

# Harpy: Run-time Code Generation in Haskell

Martin Grabmüller    Dirk Kleeblatt

Technische Universität Berlin

magr@cs.tu-berlin.de    klee@cs.tu-berlin.de

## Abstract

We present Harpy, a Haskell library for run-time code generation of x86 machine code. Harpy provides efficient generation of machine code, a convenient domain specific language for generating code and a collection of code generation combinators.

*Categories and Subject Descriptors* D.1.1 [Applicative (Functional) Programming]; D.3.4 [Processors]: Code generation

*General Terms* Languages

*Keywords* Haskell, dynamic code generation

## 1. Using Harpy

This paper describes an implementation of a run-time code generator, implemented as a Haskell library. We have made use of Haskell's features for abstraction and language extensions such as meta-programming in order to reduce the amount of boilerplate code which has to be written by the library user.

All code generation functions are based on a custom monad which maintains the state necessary for emitting binary code and for handling dynamic errors. It is also responsible for maintaining label location information e.g. for control transfer instructions. Additionally, a disassembler is provided for debugging.

*Example* As an example, let's see how Harpy can be used for generating a simple function for calculating the factorial of an unsigned 32-bit integer. The Haskell function in Fig. 1 contains the code which generates the machine code for this function. The generated function consists of a simple loop which iteratively accumulates the product up to the factorial of the parameter.

In the first two lines, two unique labels are created. One stands for the instruction which tests the loop termination condition, the other stands for the loop top. After that, the constant 1 is loaded into register `eax` and the function's parameter is loaded into `ecx`. We use Intel style assembly syntax in Harpy, i.e. first comes the destination operand, second the source operand. The jump instruction in line 5 transfers control to the end of the loop to start the first iteration. While the loop is running, `eax` accumulates the result and `ecx` is decremented until it reaches zero. The operator `@@` used in line 6 and line 8 places a label at an instruction, thus marking the beginning and the end of the loop. Lines 6 and 7 update the result and decrement the loop counter, lines 8 and 9 test for termination

```
1 fac = do loopTest <- newLabel
2   loopStart <- newLabel
3   mov eax (1 :: Word32)
4   mov ecx (Disp 4, esp)
5   jmp loopTest
6   loopStart @@ mul ecx
7   sub ecx (1 :: Word32)
8   loopTest @@ cmp ecx (0 :: Word32)
9   jne loopStart
10  ret
```

Figure 1. Code generator for factorial function

and conditionally jump back to the beginning of the loop. The function result is left in register `eax`, which is the standard result register in the C calling convention on x86 processors.

Instructions are overloaded: the `mov` instruction, for example, can be used with registers, constants or memory references as operands. This is accomplished by defining one multi-parameter type class per instruction which has the operation as its single method. The only syntactic overhead when writing code generation functions is in the creation of labels, and type annotations for overloaded numerals.

We provide a Template Haskell function `callDecl`, that can be used to define a function which calls the generated machine code. It takes the name of the function and the type of the generated code quoted by `[t | ... |]`. For our example, it looks as follows:

```
$(callDecl "callFac" [t|Word32 -> Word32|])
```

Operations in the code generation monad are executed by calling `runCodeGen`:

```
main = do
  (_,result) <- runCodeGen (fac >> callFac 8) () ()
  case result of
    Right i -> putStrLn ("fac 8 = " ++ show i)
    Left err -> putStrLn (show err)
```

After wrapping all the code fragments in this paper up with suitable imports, the example program can be compiled and run:

```
$ ghc --make -fffi -fth Main
$ ./Main
fac 8 = 40320
```

*Conclusion* Harpy has already been used successfully in two research projects: dynamic compilation for functional programs and an efficient implementation of dependent type checking.

Harpy is freely available on the Web:

<http://uebb.cs.tu-berlin.de/harpy/>