Australian National University

Mu for Functional Langauges: Retargeting the GHC Backend to a Micro Virtual Machine

Pavel Zakopaylo, ANU

Supervisors: Steve Backburn, Antony Hosking and Michael Norrish

Previous work by: Andrew Hall and Nathan Yong

Motivation

Haskell (GHC) is cross-platform, garbage-collected, concurrent.

=> Mu abstracts over hardware (=> ISA), provides GC, concurrency.

We want to demonstrate that Mu is suitable for functional programming languages.

Overview

Haskell Source

deSugar

Core Haskell

STG

codeGen

Cmm

Machine Code

Haskell Source

deSugar

Core Haskell

STG

codeGen

Cmm

Mu IR

Machine Code

Why STG?

STG → Cmm is a big transition.

=> STG is still a functional language, Cmm is a portable assembler.

Cmm code is at a similar level of abstraction to Mu IR.

But Cmm does not map well to **VM** semantics.

=> e.g. generated code hardwires the object layout, including GC metadata.

Our Aim

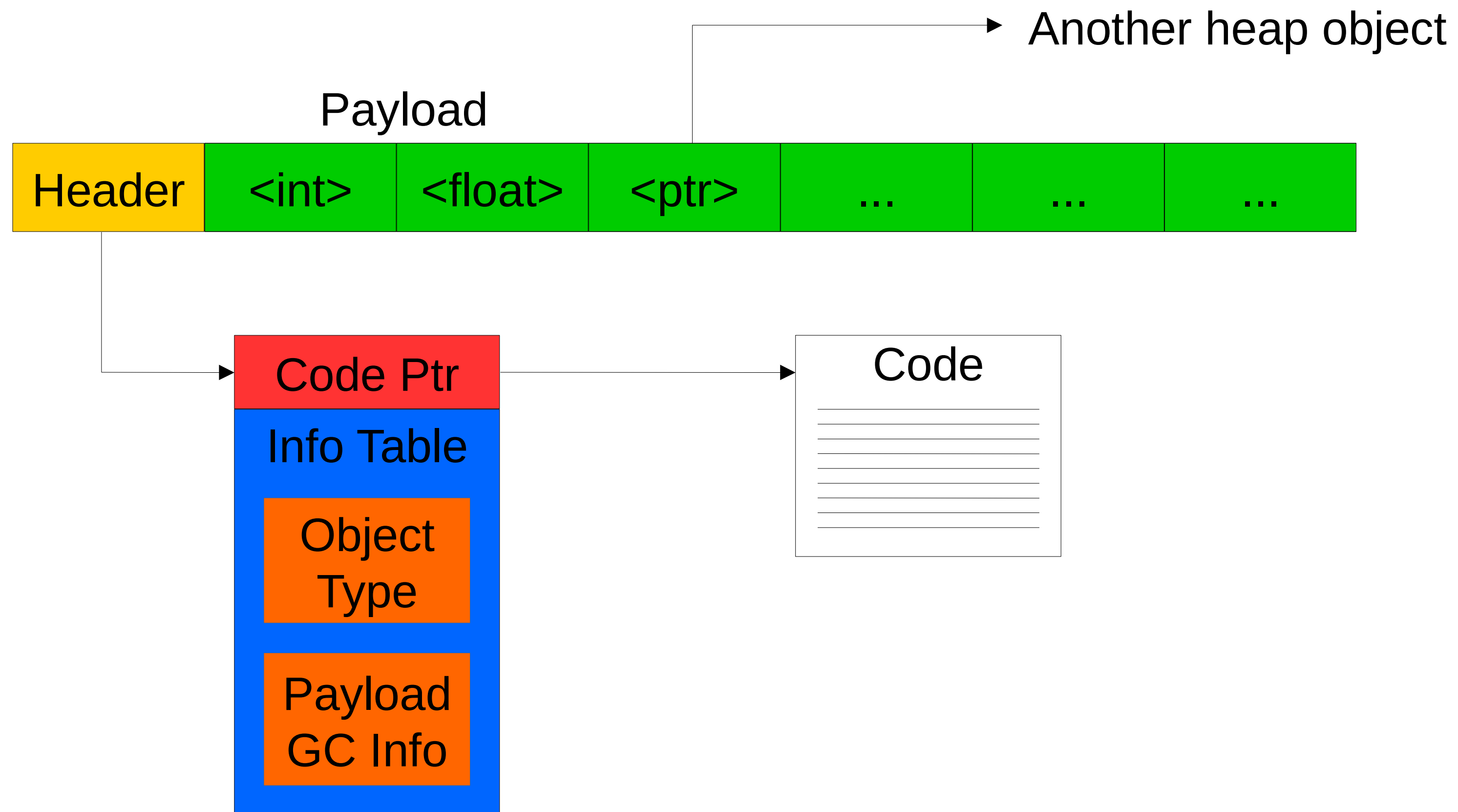
Retarget the codeGen (STG→Cmm) phase of compilation to Mu.

Use the existing infrastructure in GHC where possible.

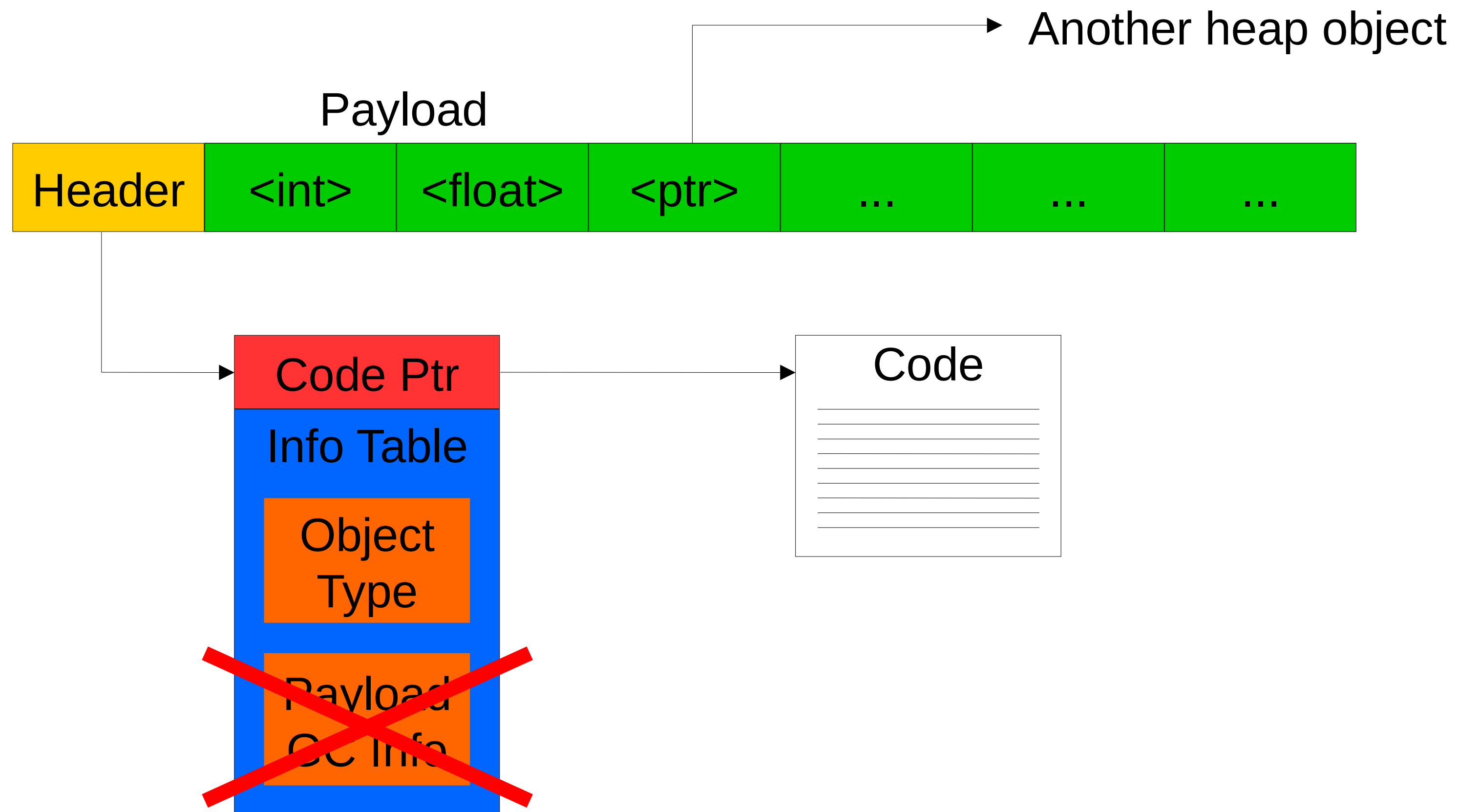
=> The compiler itself is written in Haskell.

Storage

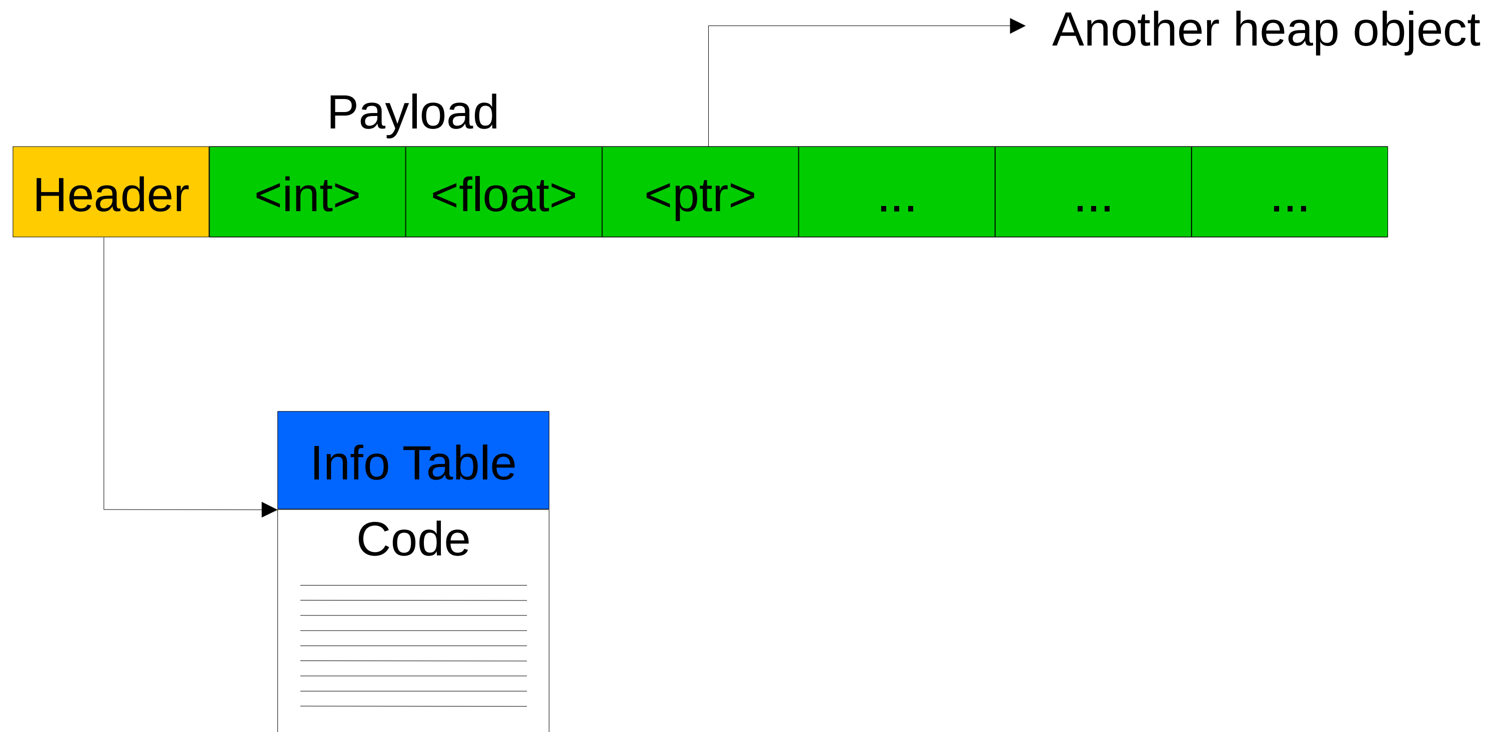
Heap Objects



Heap Objects



“Tables Next to Code”



Mu Representation

Different types of info table have different fields.

=> **Prefix rule** used so we can have references to any type of heap object anywhere.

Variable length payloads that can be made up of pointers or non-pointers.

=> **Current solution**: New Mu type for each closure. Effect on performance is currently unknown.

=> **TagRef64**: ~10 instructions for most manipulations, ints are limited to 52 bits.

=> **Unions**: Not part of Mu spec, cause issues with concurrent GC

No way to implement “tables next to code.”

Stack

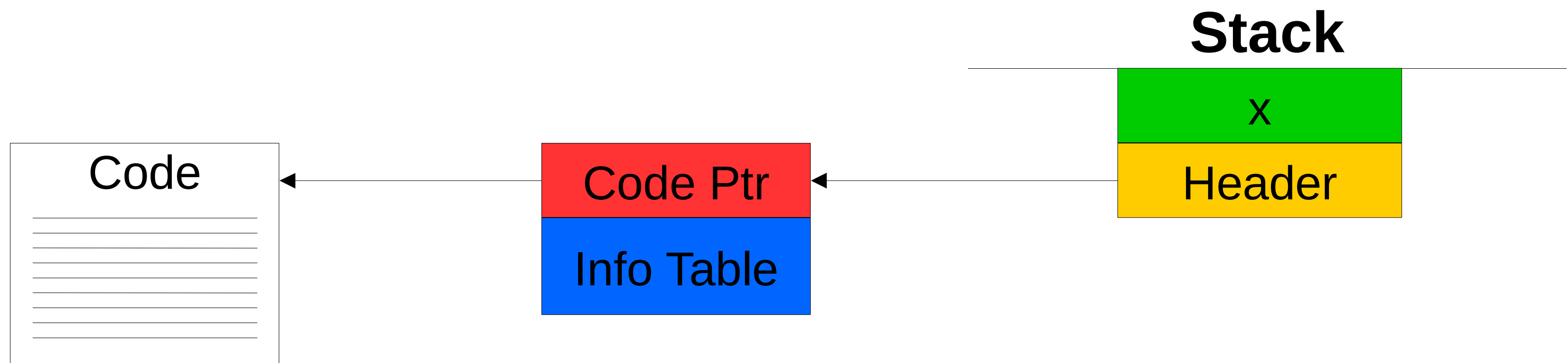
Calling convention is … unconventional.

=> Stack frames have the same layout as heap objects, where the code represents a *continuation*.

=> i.e. “Calling” a function involves pushing a stack frame and then jumping to the function’s entry point.

Stack Example

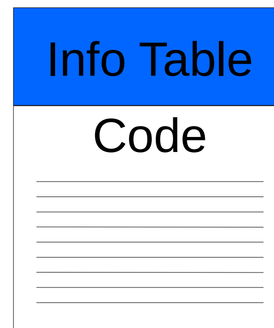
```
f :: Int -> Maybe Int -> Maybe Int
f = \x -> \y -> case y of
    Nothing -> Nothing
    Just y' -> Just (x + y')
```

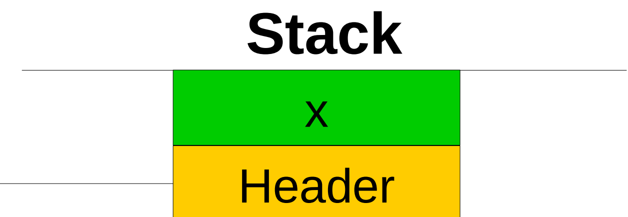


Stack Example

```
f :: Int -> Maybe Int -> Maybe Int
f = \x -> \y -> case y of
    Nothing -> Nothing
    Just y' -> Just (x + y')
```

Tables next to code ..

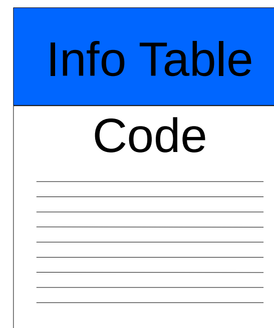


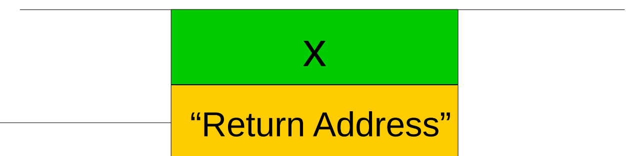


Stack Example

```
f :: Int -> Maybe Int -> Maybe Int
f = \x -> \y -> case y of
    Nothing -> Nothing
    Just y' -> Just (x + y')
```

Tables next to code ..

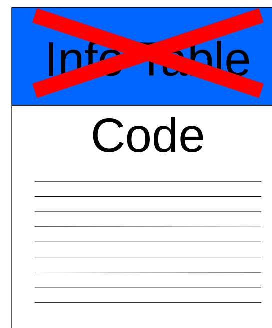


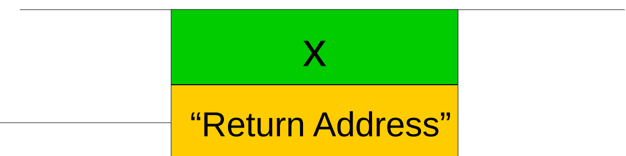


Stack Example

```
f :: Int -> Maybe Int -> Maybe Int
f = \x -> \y -> case y of
    Nothing -> Nothing
    Just y' -> Just (x + y')
```

Tables next to code ..





Australian National University

In Mu...

Let’s just use the standard calling convention!

=> Use CALL/RET instead of TAILCALL everywhere.

=> No need to roll our own stack.

… this is probably more performant w.r.t. Mu

=> x can just be a saved SSA variable.

```
f :: Int -> Maybe Int -> Maybe Int
f = \x -> \y -> case y of
    Nothing -> Nothing
    Just y' -> Just (x + y')
```

Project Status

Lots of boring but time-consuming infrastructure stuff was not covered here.

=> We can create boot images for Zebu & Holstein.

Object Layout: Fixed.

Function Applications: Partially implemented.

Case statements : Next major goal.

=> Without these nothing gets evaluated.

Project repository: https://gitlab.anu.edu.au/mu/mu-client-ghc

Summary

Haskell can use some of Mu's abstractions, notably GC.

Translating storage units into Mu is non-trivial, because GC.

It seems we can get away with changing the calling convention.

Questions?