Understanding the RealWorld

Thursday, 12 June 2014, by Edsko de Vries.
Filed under coding.

In my previous blog post Understanding the Stack I mentioned that "RealWorld tokens disappear" in the generated code. This is not *entirely* accurate. Simon Marlow gives a very brief explanation of this on the old `glasgow-haskell-users` mailing list. In this blog post we will explore this in more depth and hopefully come to a better understanding of the `RealWorld`, though perhaps not of the real world.

In order to make some points a bit easier to illustrate, we will be using a 32-bit version of ghc; and since there are no 32-bit builds of ghc for OSX later than version 7.4.2, that's what we'll use for most of this blog post.

Warmup exercise

Consider

```
constant_Int# :: () -> Int#
constant_Int# _ = 1234#
```

This compiles to the following `C--` code:

```
R1 = 1234;
Sp = Sp + 4;
```

```
    jump (I32[Sp + 0]) ();
```

On x86 architectures all Haskell function arguments are passed through the stack (on `x86-64` the first few arguments are passed through registers). That means that on entry to `constant_Int#` the top of the stack contains the `()` argument, which we discard and pop off the stack by incrementing the stack pointer `Sp`. (I will explain why I am introducing this unused argument towards the end of this blog post.)

Since `1234#` is already in weak head normal form we simply call the continuation on the top of the stack with `#1234` as the first argument. If you followed along in the previous blog post this will all be old hat to you.

Heap allocation

What happens if we use a boxed integer instead?

```
constant_Int :: () -> Int
constant_Int _ = 1234
```

If we compare the STG code for `constant_Int#` and `constant_Int` we see that the only difference is indeed that `constant_Int` must box its argument:

```
constant_Int  = \r [ds_sf9] I# [1234];
constant_Int# = \r [ds_sfb] 1234;
```

This means however that we must create a heap object, and hence the `C--` is significantly more complicated:

```
chJ:
    Hp = Hp + 8;
    if (Hp > I32[BaseReg + 92]) goto ch0;
    I32[Hp - 4] = I#_con_info;
    I32[Hp + 0] = 1234;
    R1 = Hp - 3;
```

```
        Sp = Sp + 4;
        jump (I32[Sp + 0]) ();
    chM:
        R1 = constant_Int_closure;
        jump (I32[BaseReg - 4]) ();
    ch0:
        I32[BaseReg + 116] = 8;
        goto chM;
```

The function starts with a heap overflow check. This works in much the same way as the stack overflow check that we discussed previously, so we will ignore it for the sake of this blog post, and omit heap and stack overflow checks from further code samples.

The interesting code happens after we conclude that the heap is okay. Every heap object consists of a code pointer followed by a payload. In this case, the code pointer is the one for the `I#` constructor, and the payload is `1234`.

Since `I# 1234` is already in weak head normal form, we don't need to evaluate it any further, and we can call the continuation on the stack with the address of the heap object we just constructed as its argument. Incidentally, if you are confused by the `Hp - 3`, think of it as `Hp - 4 + 1` instead; `Hp - 4` is the address of the code pointer (the start of the heap object); the `+ 1` is pointer tagging at work again.

(Incidentally, `ghc` will "preallocate" small integers, so that when you write, say, `1` instead of `1234` it doesn't actually create a new heap object but gives you a reference to a cached `Int` instead.)

Larger types

What happens if we move from `Int` to `Double`?

```
    constant_Double# :: () -> Double#
    constant_Double# _ = 1234.0##

    constant_Double :: () -> Double
```

```
constant_Double _ = 1234.0
```

The `C--` code for `constant_Double` is

```
F64[BaseReg + 56] = 1234.0 :: W64;
Sp = Sp + 4;
jump (I32[Sp + 0]) ();
```

Since a `Double` is 64-bits it cannot be passed through the 32-bit register `R1`/`esi`. On 32-bit machines `ghc` passes `Double` values through a “virtual register”. Virtual registers are just memory locations; `BaseReg` (`ebx` on `x86`) is the address of the first of them. (On `x86-64` architectures `ghc` will use the SSE registers for `Double`s.)

The code for `constant_Double` is similar to the code for `constant_Int`, except that it must allocate 12 bytes, rather than 8, for the heap object:

```
Hp = Hp + 12;
I32[Hp - 8] = D#_con_info;
F64[Hp - 4] = 1234.0 :: W64;
R1 = Hp - 7;
Sp = Sp + 4;
jump (I32[Sp + 0]) ();
```

Functions

So far we considered only constants (modulo that extra argument of type `()`). Let’s consider some simple functions next:

```
want_Int# :: Int# -> Int#
want_Int# x = x
```

translates to

```
R1 = I32[Sp + 0];
```

```
Sp = Sp + 4;
jump (I32[Sp + 0]) ();
```

Since any `Int#` is by definition already in weak head normal form, the argument to `want_Int#` must already be in weak head normal form and hence we can just pop that `Int#` off the stack and pass it to the continuation through register `R1`. The situation for `Double#` is comparable, except that as before we need to use a different register, and pop two words off the stack rather than one. That is,

```
want_Double# :: Double# -> Double#
want_Double# x = x
```

translates to

```
F64[BaseReg + 56] = F64[Sp + 0];
Sp = Sp + 8;
jump (I32[Sp + 0]) ();
```

The translation for `Int` is different.

```
want_Int :: Int -> Int
want_Int x = x
```

Since we don't know if x is in weak head normal form yet, we cannot call the continuation; instead, we must enter x itself:

```
R1 = I32[Sp + 0];
Sp = Sp + 4;
R1 = R1 & (-4);
jump I32[R1] ();
```

Remember from the previous blog post that when we enter a closure the address of the closure must be in register `R1`, in case the closure needs to lookup any free variables. So, we pop off the address of the closure from the stack, load it into `R1`, mask out any pointer tagging bits, and enter the closure. It will be the responsibility of the closure to eventually call the continuation on the stack once it reaches weak head normal form.

The translation of

```
want_Double :: Double -> Double
want_Double x = x
```

is precisely identical; indeed, both are identical to the translation of

```
id :: a -> a
id x = x
```

That's the point of boxing, of course: every argument has the same shape, and hence we can universally quantify over *any* non-primitive type. Incidentally, this is precisely why `Int#` and `Double#` have kind `#` rather than kind `*`: it means you cannot apply a polymorphic function such as `id` to an argument of a primitive type.

Calling known functions

We saw how we can *define* functions; how do we *call* them? Consider

```
call_the_want_Int# :: () -> Int#
call_the_want_Int# _ = want_Int# 1234#
```

This translates to

```
I32[Sp + 0] = 1234;
jump want_Int#_info ();
```

This couldn't be simpler. We overwrite the top of the stack (which contains the `()` argument that we are ignoring) with the argument for `want_Int#`, and then jump to the code for `want_Int#`.

For `want_Double#` the situation is slightly more complicated.

```
call_the_want_Double# :: () -> Double#
call_the_want_Double# _ = want_Double# 1234.0##
```

Since the `Double` is two words, we have to grow the stack by one word (and deal with a potential stack overflow, which we omit again):

```
F64[Sp - 4] = 1234.0 :: W64;
Sp = Sp - 4;
jump want_Double#_info ();
```

To call a function with a boxed `Int` or `Double` we have to create the heap object. For instance,

```
call_the_want_Int :: () -> Int
call_the_want_Int _ = want_Int 1234
```

translates to

```
Hp = Hp + 8;
I32[Hp - 4] = I#_con_info;
I32[Hp + 0] = 1234;
I32[Sp + 0] = Hp - 3;
jump want_Int_info ();
```

No real surprises here. We create the heap object, push the address of the newly created heap object onto the stack, and call `want_Int_info`. The same happens when we call `want_Double`, except of course that we need to create a larger heap object (12 bytes rather than 8).

Multiple arguments

Conceptually speaking Haskell does not have functions of multiple arguments. Instead, a function such as

```
want_ID## :: Int# -> Double# -> Int#
want_ID## x y = x +# double2Int# y
```

with two arguments is thought of as a function of a single argument of type `Int#`, which returns a *new* function, that takes an argument of type `Double#` and returns an `Int#`. The compiler does not implement that literally, however, as that would be much too slow. Instead, a function such as `want_ID##` really does take *two* arguments, both of which are on the stack (or in registers on architectures where that is possible); `want_ID##` gets compiled to

```
R1 = I32[Sp + 0] + %MO_FS_Conv_W64_W32(F64[Sp + 4]);
Sp = Sp + 12;
jump (I32[Sp + 0]) ();
```

We add together the `Int#` and `Double#` on the stack, store the result in `R1`, pop them off the stack and call the continuation. Similarly, when we *call* a function of multiple arguments, we push all arguments to the stack before jumping to the function entry code:

```
call_the_want_ID## :: () -> Int#
call_the_want_ID## _ = want_ID## 1234# 1234.0##
```

translates to

```
F64[Sp - 4] = 1234.0 :: W64;
I32[Sp - 8] = 1234;
Sp = Sp - 8;
jump want_ID##_info ();
```

Calling unknown functions

In the previous section we knew precisely what function we were calling. Things are more difficult if we don’t:

```
call_some_want_Int# :: (Int# -> a) -> a
call_some_want_Int# f = f 1234#
```

The reason that this is more difficult is that we do not know how many arguments `f` takes. If it takes just one, then we can call `f` in precisely the same way that we were calling `want_Int#` previously. But what if `f` takes more than one argument? In that case we must construct a `PAP` heap object to record the partial application of `f`. Clearly, the shape of this heap object must depend on the arguments that we have supplied: for an `Int#` the size of the payload must be one word, for a `Double#` it must be two, and so on.

We must also deal with the case that `f` is *already* such a PAP object. In that case, we must check if we now have all the arguments necessary to call `f`; if so, we can call `f` as we did before; if not, we must construct a new `PAP` object collecting the previously supplied arguments and the arguments we are supplying now.

Finally, if we are providing multiple arguments, we must deal with the case that `f` actually takes *fewer* arguments than we are providing and returns a new `PAP` object. In that case, we must provide all the arguments that `f` needs, call `f`, and then continue with the `PAP` object that `f` returns.

Rather than generating code to deal with all these possibilities at every call-to-unknown-function site, `ghc` delegates this to a bunch of specialized functions which are part of the RTS. The compilation of `call_some_want_Int#` therefore looks deceptively simple:

```
R1 = I32[Sp + 0];
I32[Sp + 0] = 1234;
jump stg_ap_n_fast ();
```

`stg_ap_n_fast` deals with the application of an unknown function to a single `Int#`; hence the `_n` in the name. It finds out how many arguments `f` has (by looking at the pointer tags, or failing that at the info table for `f`), and deals with all the cases that mentioned above (as well as a bunch of others; we haven't shown the full picture here). To call `stg_ap_n_fast` we pop the function argument (`f`) off the stack and store it in `R1` and then push the argument that we want to provide to `f` onto the stack.

The case for `Double#`

```
call_some_want_Double# :: (Double# -> a) -> a
call_some_want_Double# f = f 1234.0##
```

is very similar, except that we need to grow the stack by one word, and use `stg_ap_d_fast` instead (`d` for `Double#`):

```
R1 = I32[Sp + 0];
F64[Sp - 4] = 1234.0 :: W64;
Sp = Sp - 4;
jump stg_ap_d_fast ();
```

Finally, for non-primitive arguments there is a generic `stg_ap_p_fast` (`p` for pointer);

```
call_some_want_Int :: (Int -> a) -> a
call_some_want_Int f = f 1234
```

translates to

```
Hp = Hp + 8;
I32[Hp - 4] = I#_con_info;
I32[Hp + 0] = 1234;
R1 = I32[Sp + 0];
I32[Sp + 0] = Hp - 3;
jump stg_ap_p_fast ();
```

No real surprises here; we construct the heap object and call `stg_ap_p_fast`.

Multiple arguments, again

What happens if we call an unknown function with multiple arguments?

```
call_some_want_II :: (Int -> Int -> a) -> a
call_some_want_II f = f 1234 5678
```

This is really no different from supplying just one argument; we still have to deal with the same set of possibilities; the only difference is that we now need a payload for *two* pointers. This is created by `stg_ap_pp_fast`:

```
Hp = Hp + 16;
I32[Hp - 12] = I#_con_info;
I32[Hp - 8] = 5678;
I32[Hp - 4] = I#_con_info;
I32[Hp + 0] = 1234;
R1 = I32[Sp + 0];
I32[Sp + 0] = Hp - 11;
I32[Sp - 4] = Hp - 3;
Sp = Sp - 4;
jump stg_ap_pp_fast ();
```

We construct two heap objects for the two boxed integers, and then call `stg_ap_pp_fast`.

If at this point you are wondering "how many variations on `stg_ap_XYZ_fast` *are* there?" congratulate yourself, you are paying attention :) Clearly, there cannot be a version of this function for every possible number and type of argument, as there are infinitely many such combinations. Instead, the RTS only contains versions for the most common combinations. For example, there is no version for calling a function with two `Int#` arguments. So what does

```
call_some_want_II## :: (Int# -> Int# -> a) -> a
call_some_want_II## f = f 1234# 5678#
```

compile to?

```
R1 = I32[Sp + 0];
I32[Sp + 0] = 5678;
I32[Sp - 4] = I32[Lstg_ap_n_info$non_lazy_ptr];
I32[Sp - 8] = 1234;
```

```
    Sp = Sp - 8;
    jump stg_ap_n_fast ();
```

Let's consider carefully what's going on here:

1. We call `stg_ap_n_fast` (with a *single* `n`) with `1234` on the top of the stack. `stg_ap_n_fast` will notice that `f` has (at least) two arguments, and can therefore not yet be called. Instead, it creates a PAP object containing just `1234` (and the address of `f`).

2. After it has created the PAP object, it then calls the continuation on the top of the stack (after the argument). This continuation happens to be `stg_ap_n_info`, which is the "continuation wrapper" of `stg_ap_n`.

3. This in turn will pop the next argument off the stack (`5678`) and the process repeats.

In this way any non-standard version of `stg_ap_XYZ` can be simulated with a chain of standard `stg_ap_XYZ` functions.

RealWorld tokens

The main points to take away from all of the above are

- Different kinds of primitive types may have different sizes; one word for `Int#`, two for `Double#` (on a 32-bit machine).

- Different kinds of primitive types are passed through different kinds of registers; `Int#` through the standard registers, `Double#` through a virtual register (on `x86`) or through the SSE registers (on `x86-64`).

- Of course, both `Int#` and `Double#` can also be passed through the stack, in which case `Int#` takes up one word, and `Double#` takes up two.

- Similarly, the size of heap objects varies depending on the size of the primitive types.

- Finally, when calling unknown functions we need different kinds of `stg_ap_XYZ` functions to deal with different kinds of primitive types.

So what does all this have to do with the `RealWorld` tokens that I promised at the start we would look at? Well, the `RealWorld` tokens have type `State# RealWorld`, which is yet another primitive type … of size 0. So let us retrace our steps and consider the same examples that we considered for `Int#` and `Double#`, but now look at the corresponding translation for `State# RealWorld`.

We first considered the construction of a constant:

```
constant_State# :: () -> State# RealWorld
constant_State# _ = realWorld#
```

This translates to

```
Sp = Sp + 4;
jump (I32[Sp + 0]) ();
```

All we need to do is pop the `()` argument off the stack and call the continuation; since a `State# RealWorld` type has size zero, we don't need any register at all to store it!

The translation of

```
call_the_want_State# :: () -> State# RealWorld
call_the_want_State# _ = want_State# realWorld#
```

is

```
Sp = Sp + 4;
jump want_State#_info ();
```

and finally the translation of

```
call_some_want_State# :: (State# RealWorld -> a) -> a
call_some_want_State# f = f realWorld#
```

is

```
R1 = I32[Sp + 0];
Sp = Sp + 4;
jump stg_ap_v_fast ();
```

The `v` here stands for "void", and here we finally see a trace of a `RealWorld` token in the compiled code: we are applying a function to a primitive type of type void; admittedly, this is something of size zero, but it's still somehow here. Note that `stg_ap_v_fast` must still deal with all the possible cases (exact number of arguments, too many arguments, too few arguments) that we mentioned above, despite the fact that we are providing no arguments at all.

`Proxy#`

In the case of `RealWorld` and the `IO` monad this doesn't really matter most of the time. Usually when we repeatedly call an unknown IO function we are providing an argument (think `mapM`, for instance), and the STG runtime provides specialized versions of `stg_ap_XYZ_fast` for a function that takes one, two, or three pointers and a single void argument, in which case the additional void parameter does not introduce any additional overhead of the indirection through `stg_ap`. But it is good to be aware that the runtime cost is not *quite* zero when writing highly performance critical code.

However, as we start to do more and more type level programming in Haskell, we increasingly need to explicitly pass type variables around. For this reason ghc 7.8 introduces a new primitive type

```
Proxy# :: * -> #
```

with a single "constructor"

```
proxy# :: Proxy# a
```

The sole purpose of a `Proxy#` argument is to instantiate a type variable. It is used heavily for instance in the new ghc extension for overloaded records. This isn't a blog post about type level programming—quite the opposite :)—so we will just consider a completely contrived example, along the same lines of the other examples that we considered so far:

```
call_some_want_Proxies# :: (Proxy# a -> Proxy# b -> Proxy# c
                               -> (a, b, c) -> (c, b, a))
                        -> (a, b, c) -> (c, b, a)
call_some_want_Proxies# f tup = f proxy# proxy# proxy# tup
```

Although a `Proxy#` argument takes up no memory, they do play a role when calling an unknown function, as we have seen. The above function call translates to

```
R1 = P32[Sp];
I32[Sp - 12] = stg_ap_v_info;
I32[Sp - 8]  = stg_ap_v_info;
I32[Sp - 4]  = stg_ap_v_info;
I32[Sp]      = stg_ap_p_info;
Sp = Sp - 12;
call stg_ap_v_fast(R1) args: 24, res: 0, upd: 4;
```

Note the stack that we set up here: we apply the function to a single "void" argument; then when that is done, we apply it to the next, and again to the third, and only then we apply it to a "pointer" argument (the actual tuple). The use of `Proxy#` thus does incur a runtime cost. Of course, one could envision various ways in which `ghc` could be improved to alleviate this cost; the simplest of which would be to introduce some more `stg_ap_XYZ` variations targeted at `void` arguments.

For now, if this *really* matters it is faster to use a *single* `Proxy#` argument, and make sure that it is the last argument so that we can take advantage of the specialized `stg_ap` functions which were introduced for the IO monad:

```
fast_call_some_want_Proxies# :: ((a, b, c) -> Proxy# (a, b, c) -> (c, b
                             -> (a, b, c) -> (c, b, a)
fast_call_some_want_Proxies# f tup = f tup proxy#
```

compiles to just

```
    R1 = P32[Sp];
    Sp = Sp + 4;
    call stg_ap_pv_fast(R1) args: 8, res: 0, upd: 4;
```

Footnotes

What is `I#_con_info`?

We saw that the heap object header for boxed integers is `I#_con_info`. You might be wondering what exactly *is* at that code pointer address. We can fire up `lldb` (or `gdb`) and find out:

```
(lldb) disassemble -n ghczmprim_GHCziTypes_Izh_con_info
Main`ghczmprim_GHCziTypes_Izh_con_info:
Main[0x68398]:  incl   %esi
Main[0x68399]:  jmpl   *(%ebp)
```

`ghczmprim_GHCziTypes_Izh` is the Z-encoding for `ghc-prim_GHC.Types.I#`; `zm` for minus, `zh` for hash, and `zi` for dot ("the dot on the i"). So what does the code do? Well, the code pointer associated with every heap object is the code that reduces that heap object to normal form (this what we mean by "entering a closure"). Since a constructor application already *is* in normal form, there is almost nothing to do, so we just call the continuation on the stack (`jmpl`).

The only complication is due to pointer tagging, once again. Remember that when we evaluate a closure, `R1` (mapped to `esi` on x86 architectures) points to the address of the closure. If we entered the closure that means the pointer wasn't tagged yet, or we ignored the tag bits (for whatever reason); we make sure to tag it by calling `incl` before calling the continuation so that we don't unnecessarily enter the closure again.

Info tables

The discussion of `constant_Int` above assumed that the "tables next to code" optimization is enabled (which is most of the time). With this optimization enabled the code pointer for a closure also doubles as a pointer to the *info table* for the closure.

The info table for a closure contains information such as the constructor tag (which constructor of a data type is this?), the size of the payload and the types of its elements (int, float, pointer etc) for garbage collection purposes, the static reference table (used for garbage collection of CAFs), profiling information, etc. The info table is located just *before* the closure entry code in memory.

When the optimization is *not* enabled the heap object header is a pointer to the info table, which in turn contains a field for the entry code. This means that in order to enter a closure an additional indirection is necessary.

Allocation in STG

I have often seen claims that the *only* place where heap allocation occurs in STG code is in `let` expressions (for instance, on StackOverflow and on cvs-ghc). This is not entirely accurate, as we saw when we looked at `constant_Int`. See `cgExpr` in `compiler/codeGen/StgCmmExpr` in the ghc sources, and in particular function `cgConApp` under “tail calls”.

Boxing state

We don’t normally box `State#` objects, but we could:

```
data State a = State# (State# a)

constant_State :: () -> State RealWorld
constant_State _ = State# realWorld#
```

The translation of `constant_State` is however a bit surprising:

```
Hp = Hp + 8;
if (Hp > I32[BaseReg + 92]) goto cm8;
I32[Hp - 4] = State#_con_info;
R1 = Hp - 3;
Sp = Sp + 4;
jump (I32[Sp + 0]) ();
```

This creates a heap object with a one-word payload, which is subseqeuently left unused. The reason is that every heap object must be at least two words, so that there is enough room to overwrite it with a forwarding pointer during garbage collection. Thanks to `rwbarton` on reddit for pointing this out!

That strange `()` argument

Many of the examples we looked at had an additional `()` argument:

```
call_the_want_Int# :: () -> Int#
call_the_want_Int# _ = want_Int# 1234#
```

What is the purpose of this argument? Well, first of all, a top-level binding to a primitive type (something of kind #) is not allowed; this is illegal Haskell:

```
caf_the_want_Int# :: Int# -- Not legal Haskell
caf_the_want_Int# = want_Int# 1234#
```

But even for examples where it would be legal we want to avoid it for the sake of this blog post. Consider

```
caf_the_want_Int :: Int
caf_the_want_Int = want_Int 1234
```

The difference between `call_the_want_Int` and `caf_the_want_Int` is that the latter is a CAF or "constant applicative form"; `ghc` must ensure that `caf_the_want_Int` is reduced to weak head normal form only once in the execution of the program. CAFs are a large topic in their own right; another blog post, perhaps (for now, some interesting links are this Stack overflow question and ghc ticket #917, answer from Simon Peyton-Jones about how CAFs are garbage collected, and the entry on the ghc wiki). If you are feeling brave, take a look at the generated code for `caf_the_want_Int`.

In order to avoid CAFs being generated, we introduce the additional argument, and use the compiler flag `-fno-full-laziness` to make sure that `ghc` doesn't attempt to optimize that additional argument away again.