Types (are/want to be) Calling Conventions

Ben Lippmeier, LambdaJam, 2019/5/14

Binding Arity

```
f0 : Nat -> Nat -> Nat
f0 = λx. λy. x + y                (arity 0)

f1 : Nat -> Nat -> Nat
f1 x = λy. x + y                  (arity 1)

f2 : Nat -> Nat -> Nat
f2 x y = x + y                    (arity 2)
```

These bindings all have the same type,
but different *binding arity*.

Now with Type Parameters

```
g0 : ∀a. Bool -> a -> a -> a
g0 = Λa. λb:Bool. λx:a. λy:a      (arity 0,0)
   . if b then x else y
```

```
g1_2 : ∀a. Bool -> a -> a -> a
g1_2 @a (b:Bool) (x:a)              (arity 1,2)
   = λy:a. if b then x else y
```

Trouble...

```
g0 : Bool -> ∀a. a -> a -> a
g0 = λb:Bool. Λa. λx:a. λy:a      (arity 0,0)
   . if b then x else y
```

`g0` has a rank-1 type but it is not in “prenex form”

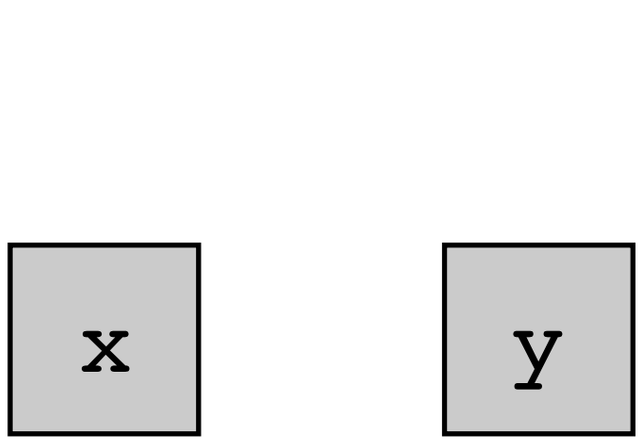
```
gXXX : Bool -> ∀a. a -> a -> a
gXXX (b:Bool) @a (x:a)              (arity ???)
   = λy:a. if b then x else y
```

Parameter and Return Vectors

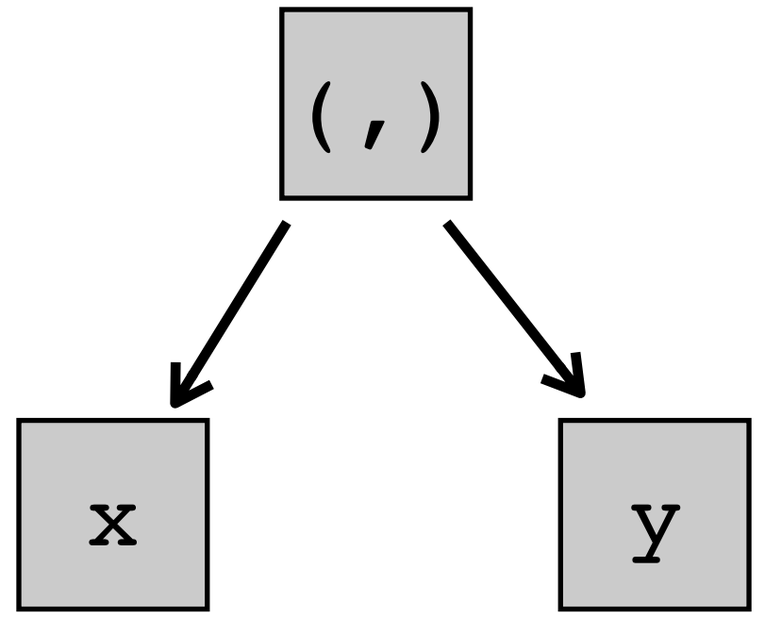
```
f2 : [Nat, Nat] :-> [Nat]
f2 [x, y] = [x + y]
```

Parameter and Return Vectors

```
f2 : [Nat, Nat] :-> [Nat]
f2 [x, y] = [x + y]
```



[x, y]
"vector"



(x, y)
"tuple"

Parameter and Return Vectors

```
f2 : [Nat, Nat] :-> [Nat]
f2 [x, y] = [x + y]

g2 : [Nat] :-> [Nat, Nat]
g2 [x] = [x, x]
```

Parameter and Return Vectors

```
f2 : [Nat, Nat] :-> [Nat]
f2 [x, y] = [x + y]

g2 : [Nat] :-> [Nat, Nat]
g2 [x] = [x, x]
```

```
f1 : [Nat] :-> [[Nat] :-> [Nat]]
f1 [x] = [λ[y]. [x + y]]
```

Parameter and Return Vectors

```
f2 : [Nat, Nat] :-> [Nat]
f2 [x, y] = [x + y]

g2 : [Nat] :-> [Nat, Nat]
g2 [x] = [x, x]
```

```
f1 : [Nat] :-> [[Nat] :-> [Nat]]
f1 [x] = [λ[y]. [x + y]]

f0 : [] :-> [[Nat] :-> [[Nat] :-> [Nat]]]
f0 [] = [λ[x].[λ[y]. [x + y]]]
```

Polymorphism

```
gXX : Bool -> ∀a. a -> a -> a
gXX (b:Bool) @a (x:a)
 = λy:a. if b then x else y
```

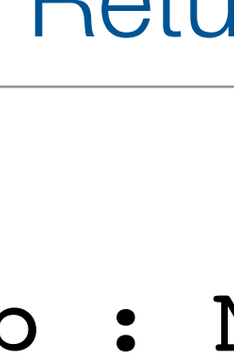
"forall"

```
gYY : [Bool] :-> [[a] :*> [[a, a] :-> [a]]]
gYY [b:Bool]
  = [Λ[a]. [λ[x,y]. if b then [x] else [y]]]
```

Multiple Return Values vs Tuples

```
fTup : Nat -> (Nat, Nat)
fTup x = (x, x)

gUse : Bool -> Nat -> (Nat, Nat)
gUse b x
 = let z = fTup x
   in if b then z else (0, 0)
```



A single variable binds a tuple containing two values. This implies we have allocated a container object.

All returned values must be bound explicitly

```
fVec : [Nat] :-> [Nat, Nat]
fVec [x] = [x, x]
```

When it returns two values..

```
gUse : [Bool, Nat] :-> [Nat, Nat]
gUse [b, x]
 = let [z1, z2] = fVec [x]
   in  if b then [z1, z2] else [0, 0]
```

.. we must *bind* two values

.. and there is no intermediate allocation.

Parameter/Return vectors vs GHC UnboxedTuples

This works in GHC

```
f x y = (# x+1, y-1 #)
g x   = case f x x of
          { (# a, b #) -> a + b }
```

.. but this does not:

```
g :: (# Int, Int #) -> Int
g (# a, b #) = a
```

... as it wants to be sugar for:

```
g :: (# Int, Int #) -> Int
g x = case x of { (# a, b #) -> a }
```

salt
\discus-lang
\github.com

Discus

Functional,
with implicit allocation

Salt

Functional + Procedural
with explicit allocation.

(abstract)
Machine Instructions

LLVM

```c
void hello(void) {
  printf("hello world\n");
}
```

C

```c
void hello(void) {
  printf("hello world\n");
}
```

C

```haskell
data Void

hello :: IO Void
hello
  = do putStrLn "hello world"
       ... ?
```

Haskell

```c
void hello(void) {
  printf("hello world\n");
}
```

C

```haskell
data Void

hello :: IO Void
hello
  = do putStrLn "hello world"
       ... ?
```

Haskell

```
hello : Console ! []
 = do putStrLn "hello world"
      []
```

Salt

```
-- Functional Factorial
term fac1 [x: #Nat]: #Nat
 = if    #nat'eq [x, 0]
   then 1
   else #nat'mul [ x
                 , fac1 [#nat'sub [x, 1]]]
```

```
-- Procedural Factorial
proc fac2 [n: #Nat]: #Nat
 = do   cell acc: #Nat ← n
        cell x:   #Nat ← n
        enter fac []
         with fac []: [] = do
              when #nat'eq [x, 0] do leave
              acc ← #nat'mul [acc, x]
              x   ← #nat'sub [x,   1]
              fac []
      yield acc
```

Types are calling conventions

Max Bolingbroke
University of Cambridge
mb566@cam.ac.uk

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Abstract

It is common for compilers to derive the calling convention of a function from its type. Doing so is simple and modular but misses many optimisation opportunities, particularly in lazy, higher-order functional languages with extensive use of currying. We restore the lost opportunities by defining Strict Core, a new intermediate language whose type system makes the missing distinctions: laziness is explicit, and functions take multiple arguments and return multiple results.

1. Introduction

In the implementation of a lazy functional programming language, imagine that you are given the following function:

$$f :: Int \to Bool \to (Int, Bool)$$

How would you go about actually executing an application of f to two arguments? There are many factors to consider:

In this paper we take a more systematic approach. We outline a new intermediate language for a compiler for a purely functional programming language, that is designed to encode the most important aspects of a function's calling convention directly in the type system of a concise lambda calculus with a simple operational semantics.

- We present Strict Core, a typed intermediate language whose types are rich enough to describe all the calling conventions that our experience with GHC has convinced us are valuable (Section 3). For example, Strict Core supports uncurried functions symmetrically, with both multiple arguments and multiple results.
- We show how to translate a lazy functional language like Haskell into Strict Core (Section 4). The source language, which we call FH, contains all the features that we are interested in compiling well – laziness, parametric polymorphism, higher-order functions and so on.

In Haskell Symposium'2009