Types (are/want to be) Calling Conventions

Ben Lippmeier
Digital Asset
FP-Syd 2019/1/23

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*.

Types want to be Calling Conventions

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

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

To a compiler backend these are different functions,
need different code to be generated,
and should rightly have different types.

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 = λx: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”

```
gXX : Bool -> ∀a. a -> a -> a
gXX (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]

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

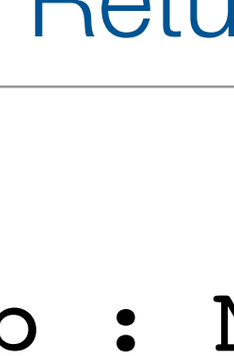
```
f1 : [Nat] :-> [[Nat] :-> [Nat]]
f1 [x] = [λ[y]. [x + y]]

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

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.

Polymorphism

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

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

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 }
```

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

Search or jump to... / Pull requests Issues Marketplace Explore

discus-lang / salt

Watch 7 | Star 32 | Fork 2

Code | Issues 1 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

The compilation target that functional programmers always wanted. Edit

language functional low-level compiler compilation target lambda-calculus Manage topics

281 commits | 1 branch | 0 releases | 3 contributors

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

benl23x5 core: be consistent about default branches across if/case term and pr... Latest commit a1fc900 4 hours ago

bin	Adding bin/ directory and dummy file to keep it.	3 months ago
doc	core: be consistent about default branches across if/case term and pr...	4 hours ago
make	make: go to XStrict by default	4 days ago
src	core: be consistent about default branches across if/case term and pr...	4 hours ago
test	core: more proc parsing, and pull out 'else' branch of term 'if' so i...	5 hours ago
.gitignore	add stack.yaml to get intero/ghcid support	20 days ago
Makefile	Property tests: "make waves"	2 months ago
README.md	readme: copy edits	4 days ago
package.yaml	make: go to XStrict by default	4 days ago
salt.cabal	more readme	6 days ago
stack.yaml	add stack.yaml to get intero/ghcid support	20 days ago

README.md

Salt Intermediate Language

Salt is what you get when you leave C out in the sun for too long.

Salt is the compilation target that functional programmers always wanted.

Salt is a System-F variant intended as an intermediate language between higher level languages and an abstract assembly language (LLVM). Hand written code can also be used to implement runtime systems. The Disco Discus Compiler uses an earlier version of Salt (v1), and its runtime system is written in it. This current repo contains a newer version of Salt (v2) which is being split out into its own project. Salt v1 has a working LLVM backend, but the one for v2 in this repo is still a work in progress.

Example

We have some small "Hello World" type examples and test cases, but no larger programs yet. See the demo and syntax directories in this repo. The syntax looks like:

```
term reverse @[a: #Data] [xx: #List a]: #List a
 = case #list'case @a xx of
        { nil  [] → [list a|]
```