Note of Implement Functional Languages

Hong

August 31, 2017

Contents

1 Preface **2**

1.1 Overview of the implementations 2

2 Chaptter 1: The Core Language **3**

2.1 Overview of the core language 3

2.1.1 Local definitions 3

2.1.2 Structured data types 3

2.1.3 Representing constructors 4

2.1.4 `case` expressions 4

2.2 Syntax of the Core Language 4

2.3 Structured Data Types 5

2.4 A parser for the Core Language 7

1 Preface

- understand the implement of non-strict functional language
  [lazy graph reduction]

- make functional language implementations “come alive”

1.1 Overview of the implementations

1. source code
2. praser [Chapter 1]
3. Lambda lifter [Chapter 6]
4. core program
5. - Template compiler → Template interpreter [Chapter 2]
   - G-machine compiler → G-machine interpreter [Chapter 3]
   - TIM ... [Chapter 4]
   - Parallel G-machine ... [Chapter 5]

interpreter. The compiler takes a Core-language program and translates it into form suitable for execution by the machine interpreter.

lambda lift. It turns local function definitions into global ones, thus enabling local function definitions to be written freely and transformed out later.

2 Chaptter 1: The Core Language

2.1 Overview of the core language

Core program consists of a set of *supercombinator definitions*, in which **main** is a distinguished one.
Not all supercombinators have arguments. Some, such as **main**, take no arguments, which are called *constant applicative forms* or CAFs.

```
main = double 21
double x = x + x
```

2.1.1 Local definitions

Supercombinators can have local definitions:

```
main = quadruple 21
quadruple x = let double_x = x + x
              in  double_x + double_x
```

A let expression is *non-recursive*, use **letrec** for recursive definitions.
Local functions and pattern matching are not provided by Core language **let** and **letrec** executions.

Local function can only be defined at the top level;
Pattern matching — **case** expressions.

2.1.2 Structured data types

algebraic data types

```
colour = Red | Green | Blue
tree a = Leaf a | Branch (tree a) (tree a)
```

Structured values are *built* with constructors, and *taken apart* using *pattern matching*

How are we to represent and manipulate structured types in small core language?

- Use a simple, uniform representation for all constructors
- Transforme pattern matching into simple case expressions

2.1.3 Representing constructors

$$\texttt{Pack}\{tag, arity\}$$

- *tag* is an integer to uniquely indentify the constructors
- *arity* tells how many arguments it takes

2.1.4 `case` expressions

```
-- t is a tree
depth t = case t of
                <1> n -> 0
                <2> t1 t2 -> 1 + max (depth t1) (depth t2)
```

`case` expressions: evaluate the expression to be analysed, get the tag of the constructor it is built with and evaluate the appropriate alternative.

2.2 Syntax of the Core Language

Precedence	Associativity	Operator
6	Left	Application
5	Right	*
	None	/
4	Right	+
	None	-
3	None	== == ¿ ¿= ¡ ¡=
2	Right	&
1	right	—

2.3 Structured Data Types

```haskell
moudle Language where
    data Expr a = Evar Name                             -- 变量
                | ENum Int                              -- 数值
                | Econstr Int Int                       -- 构造函数
                | EAp (Expr a)  (Expr a)                -- 函数应用
                | ELet IsRec [(a, Expr a)] (Expr a)     -- let 定义 IsRec 是否能递归定义 [(a, Expr a)] 定义 Expr
                | ECase (Expr a) [Alter a]              -- case 表达式
                | ELam [a] (Expr a)                     -- lambda 表达式
                deriving (show)

    type CoreExpr = Expr Name
```

Figure 1: structured data types

```
x + y  -> EAp (EAp (EVar) "+") (EVar "x")) (EVar "y")
```

Definitions:

```haskell
> type Name = String
>
> type IsRec = Bool
> recursive, nonRecursive :: IsRec
> recursive    = True
> nonRecursive = False
>
> bindersOf :: [(a, b)] -> [a]  -- Pack(tag, arity)
> bindersOf defns = [name | (name, rhs) <- definitions]
> rhssOf :: [(a, b)] -> [b]
> rhssOf defns = [rhs | (name, rhs) <- defns]
>
> -- case
> type Alter a = (Int, [a], Expr a)
> type CoreAlt = Alter Name
>
> isAtomicExpr :: Expr a -> Bool
> isAtomicExpr (EVar v) = True
> isAtomicExpr (ENum n) = True
> isAtomicExpr _        = False
>
> type Program a = [ScDefn a]  -- ScDefn: supercombinator definitions
> type CoreProgram = Program Name
> type ScDefn a = (Name, [a], Expr a)  -- [a]: argument list Expr
> type CoreScDefn = ScDefn Name
```

A small Example:

```
main = double 21
double x = x + x
```

This program is represented by the following Haskell expression, of type `CoreProgram`

```
[
    ("main",    [],     (EAp (EVar "double") (ENum 21))),
    ("double",  ["x"],  (EAp (EAp (EVar) "+") (EVar "x")) (EVar "x")))
]
```

prelude definitions

```
preludeDefs :: CoreProgram
preludeDefs = [
    ("I", ["x"], EVar "x"),
    ("K", ["x", "y"], EVar "x"),
    ("K1", ["x", "y"], EVar "y"),
    ("S", ["f", "g", "x"], EAp (EAp (EVar "f") (EVar "x"))
                               (EAp (EVar "g") (EVar "x"))),
    ("compose", ["f", "g", "x"], EAp (EVar "f")
                                     (EAp (EVar "g") (EVar "x"))),
    ("twice", ["f"], EAp (EAp (Evar "compose") (EVar "f")) (EVar "f"))
]
```

Programs	*program*	$\rightarrow$	$sc_1; ...; sc_n$	$n \geq 1$
Supercombinators	*sc*	$\rightarrow$	$var\ var_1 ... var_n\ =\ expr$	$n \geq 0$
Expressions	*expr*	$\rightarrow$	$expr\ aexpr$	Application
		$\mid$	$expr_1\ binop\ expr_2$	infix binary Application
		$\mid$	`let` *defns* `in` *expr*	Local definitions
		$\mid$	`letrec` *defns* `in` *expr*	Local recursive definitions
		$\mid$	`case` *expr* `of` *alts*	Case expressions
		$\mid$	$\lambda var_1 ... var_n\ .\ expr$	Lambda abstraction $n \geq 1$
		$\mid$	*aexpr*	Atomic expression
	aexpr	$\rightarrow$	*var*	Variable
		$\mid$	*num*	Number
		$\mid$	`Pack`$\{num, num\}$	Constructor
		$\mid$	$(expr)$	Parenthesised expression
Definitions	*defns*	$\rightarrow$	$defn_1; ...; defn_n$	$n \geq 1$
	defn	$\rightarrow$	$var = expr$	
Alternatives	*alts*	$\rightarrow$	$alt_1; ...; alt_n$	$n \geq 1$
	alt	$\rightarrow$	$< num >\ \ var_1 ... var_n - > expr$	

2.4 A parser for the Core Language

Read a file containing the Core program in its concrete syntaxm and parse it to a value of type `CoreProgram`

- Obtain the contents of the named file as a list of characters [built-in Haskell function `read`]
- *lexical analysis* function `lex` breaks the input into a sequence of small chunks, such as indentifies, numbers, symbols..., which are called *tokens*

```
> clex :: String -> [Token]
```

- Finally, the *syntax analysis* function `syntax` consumes this sequence of tokens and produces a `CoreProgram`

```
> syntax :: [Token] -> CoreProgram
```