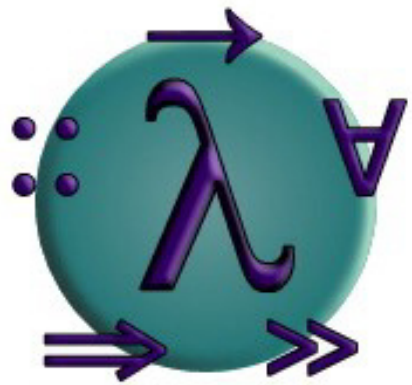
Compiling Functional Programs in a Statically-typed Language



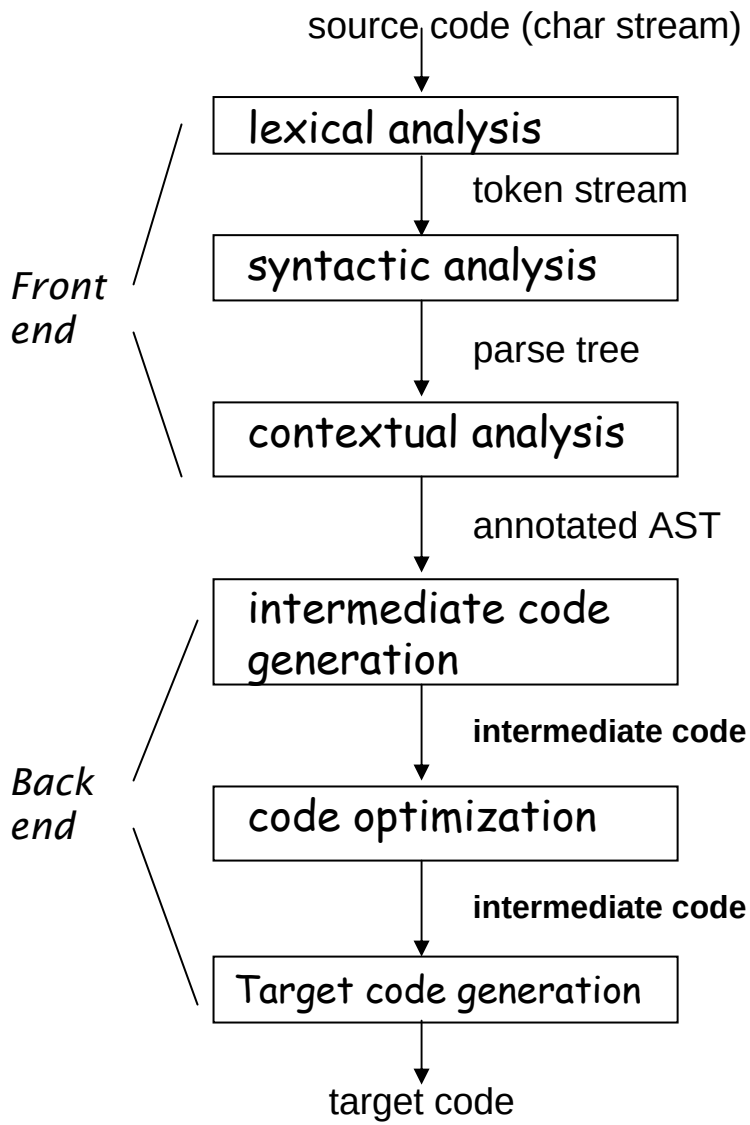
Xiao-lei Cui
Nov 24th, 2006

Compilation in General

- In a broader view, a compiler is a program which processes a structured source and generates (simpler structured) target code. Compilation is typically split into a number of consecutive phases.

- Source code: programming language, text formatting language, database query language, etc.
- Target code: high-level language, assembly language, virtual-machine code, machine code, etc.

- Front end: consists of the phases serve to figure out the *meaning* of the source code; operations are mostly dependent to the source language.
- Back end: consists of the phases serves to construct target code; operations are mostly dependent to the target architecture.



Outline

Section 1: A Short Tour of Haskell
(review essential functional programming aspects)

Section 2: Front End of a functional compiler
(address on aspects handled at the Front End)

Section 3: Back End of a functional compiler
(address on aspects handled at the Back End)

Short Tour of Haskell..

- Compiling functional programs differs considerably from compiling imperative programs. Before discussing techniques employed in a functional language compiler, we quickly walk through a tour of Haskell to highlight the aspects that require special care.

- We will emphasize the aspects of functional languages that raise the level of abstraction above that of imperative languages.

- Haskell:
  - representative of "pure functional lazy language"
  - contains most features from functional programming
  - statically-typed: Haskell is strongly-typed and the type-checking can be performed at compile-time.

- Offside Rule

Function is defined in the form of an equation. There is no explicit token to denote the end of each equation. The Offside rule controls the bounding box of an expression (the function name).

Offside Rule states:

Anything belonging to a function definition must be indented from where that function definition began; anything that is not indented is taken to be a new function definition.

Example:

```
twice x = 2 * x
```

⇕

```
twice x =
 2 * x
```

but not same as:

```
twice x =
2 * x
```

- Offside Rule (continued)
  - The Offside rule has to be applied recursively for nested function definitions, as in:

```
fac n = if (n == 0) then 1 else prod n (n-1)
      where
        prod acc n = if (n == 0) then acc
                     else prod (acc*n) (n-1)
```

Handled conveniently by lexical analyzer (How?)

Lexical analyzer inserts explicit end-of-equation token, a ";" in Haskell, as follows.

It maintains a stack of offside markers.

Step1: When the lexical analyzer detects the first character of a function name, it pushes the cursor position of this character on stack. Upon detecting a line-break, it skips all white space and records the first character position of the next function name.

Step2: It then compares this new position with the top of the marker stack. If the new position is less or equal to the marker, it pops the marker and inserts a ";" token; the lexical analyzer then compares the new position with the top of the stack repeatedly, until the stack is empty OR the new position is greater than the top marker. Continue with Step 1.

Exercise:

According to the Offside rule in functional programming, if the entire program has been scanned (lexical analysis completes), can you tell anything about the state of the marker stack ?

- empty
- non-empty

- List

`[]`	empty list
`[1]`	list of integer
`[1,2,3]`	...
`["red", "green", "blue"]`	list of string
`[1..5]`	arithmetic sequence shorthand for `[1,2,3,4,5]`

Note:

An arithmetic sequence ([n..m]) can be constructed by a function as follows

```
range n m = if n>m then []
            else (n : range (n+1) m)
```

- List Comprehension

In mathematics, the comprehension notation can be used to construct a new sets from old sets.

$$\{x^2 \mid x \in \{1..5\}\}$$

In Haskell, a similar comprehension notation can be used to construct new lists from old lists.

```
[x^2 | x <- [1..5] ]
```

List Comprehension is considered to be syntactic sugar since it can be transformed easily to a simpler expression. This transformation will be discussed in Section 2.

- Pattern matching

  - In general a function can be specified by several equations containing patterns at argument position.
  - A pattern can be a constant, a variable or a constructor whose elements are patterns such as (x:xs).
  - Function definitions based pattern matching can be translated to equivalent definitions based on if-then-else construct. Further details will be shown in Section 2.

Example:

```
length []     = 0
length (x:xs) = 1 + length xs
```



```
length list =
        if (list == [])  then 0
        else let
                x  = head list
                xs = tail list
             in
                1 + length xs
```

- Higher-order function
  - A function that takes a function as an argument, or delivers one as a result.
  - Currying allows us to create new functions by partially apply an existing function to arguments that is less than the arity of the function.
  - Functional languages consider a function with n argument as syntactic sugar for a chain of n unary functions processing the arguments one by one:

    $$f\ e_1\ e_2\ \ldots\ e_n\ ==\ (\ldots\ ((f\ e_1)\ e_2)\ \ldots\ e_n)$$

  - In short, an n-ary function applied to m arguments denotes an (n-m)-ary function. A 0-ary function can be viewed as an expression.
  - We need additional run-time support to construct and evaluate curried functions. This will be discussed later in Section 3.

- Lazy Evaluation
  - Evaluate a non-trivial expression amounts to evaluating its sub-expressions and combining the results by operators or functions.
  - Lazy Evaluation specifies a subexpression will only be evaluated when its value is needed for the progress of the computation.
  - Example:

```
ignorearg x = "I did not evaluate the arg."
```

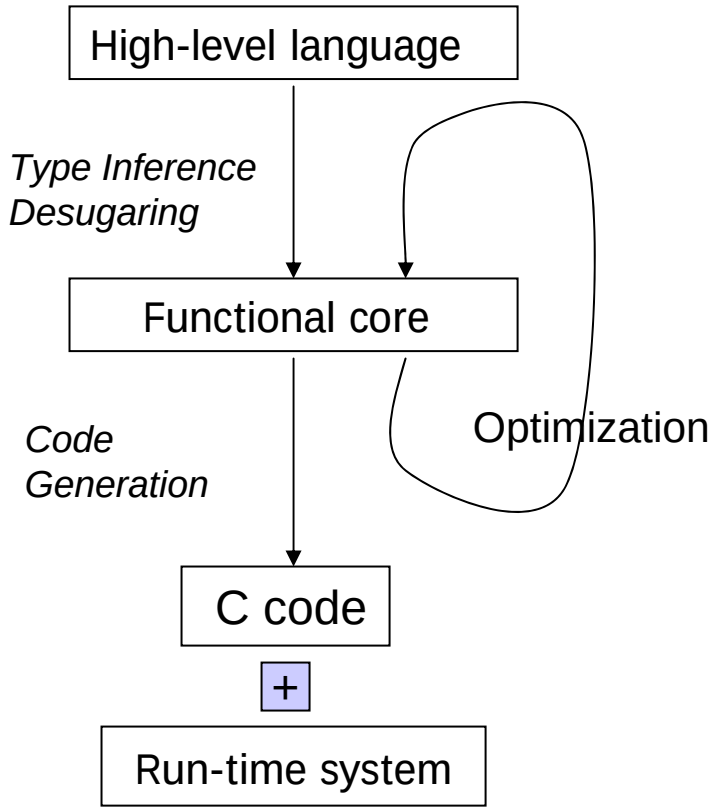
```
> ignorearg (1/0)
"I did not evaluate the arg.“

> seq ignorearg (1/0)
1.#INF
```

- Lazy evaluation requires run-time support. A function application is translated into a graph containing function and arguments. When needed, such a suspended function can be activated at run-time.

Front End of the compiler..

- General structure of a functional compiler is given by the picture to the right.
- High-level source code is simplified through several transformations into a functional core (intermediate code in Haskell), which contains no syntactic sugar.
- The Functional core requires explicit typing, therefore the front end has to derive type information from source code.

  (type checking is done at compile-time)
- Optimizations may be applied to functional core multiple times.
- In the final step, code generation transform functional core to target code (C code); the resulting C program is then linked with the run-time system and compiled with a C compiler.



..Front End of the compiler..

- Functional core

  It includes the following constructs:

  - Basic data types
  - Structured data types
  - Typed non-nesting functions
  - Local bindings
  - Expression consisting of identifier, arithmetic operator, if-then-else compound, and function application
  - Higher-order functions
  - Lazy evaluation semantics

  These constructs, except for the last two, can be easily mapped to C contructs.

- Polymorphic type checking

  - Type checker must derive type information solely from function definitions.
  - When type checking an n-ary function, n fresh types Typ_i are added for each argument.
  - Setting up a set of equations (the constraint), and then solve (unify) the equations.
  - See the Hindley-Milner Type Inference Algorithm
    - Chapter 22 in "Type and Programming Languages" [Pierce]
    - Chapter 16 in "Modern compiler implementation in ML" [Appel]

..Front End of the compiler..

- Desugaring (removing syntactic sugar)
  - After type checking, the compiler performs Desugaring.
  - Focus on translating:
    - List
    - List comprehension
    - Pattern matching
    - Nested function

    into their equivalent functional core constructs.

1. List

   List notations contain 3 forms of syntactic sugar:

   , : ..

[1,2]	x : xs
➔ Cons(1,Cons(2,Nil))	➔ Cons(x, xs)

[n..m] ➔

```
range n m = if n>m then [ ]
                   else (n : range (n+1) m)
```

2. List comprehension

The general form of list comprehension

[expression | qualifier, ... , qualifier]

where a *qualifier* is either a generator of a filter(F):

generator ::= var <- list comprehension

filter ::= boolean expession

Example in Haskell

```
pyth n = [(a,b,c) | a <- [1 .. n],
                    b <- [a .. n],
                    c <- [b .. n],
                    a^2 + b^2 == c^2]
```

List comprehension continued

function *pyth* n computes all Pythagorean triangles with sides less than or equal to n.

- Transformation of list comprehensions works by processing the qualifiers from left to right one at a time.
- Translation scheme for list comprehension is the following (F denotes filter; Q denotes a sequence of qualifiers; L denotes a list expression):

T{ [*expr* |] } ➔ [*expr*] (1)

T{ [*expr* | *F*, *Q*] } ➔ if (*F*) then T{ [*expr* | *Q*] } (2)
else [ ]

T{ [*expr* | *e* <- *L*, *Q*] } ➔ mappend f$_Q$ *L* (3)
where
f$_Q$ *e* = T{ [*expr* | *Q*] }

- Rule (1) covers the base case; no more qualifiers left.
- Rule (2) handles the filter qualifier(F). If F holds, we compute the remaining Q by recursively invoking translation scheme T; otherwise, return empty list and terminates.
- Rule (3) covers the generator qualifier, e <- L. Since the generator produces zero or more element drawn from L, we need generate code to iterate over all elements in L, compute the remainder Q, and concat result lists to a single list.
  - Need a nested function fQ. It takes element e and produces a list of values that Q can assume for e.
  - Need a function that concat the result lists into a single list, *mappend* (similar to List.map):

```
mappend :: (a -> [b]) -> [a] -> [b]
mappend f [ ]     = [ ]
mappend f (x:xs) = f x ++ mappend f xs
```

Applying the translation scheme to function *pyth* produces code:

```
pyth n = mappend f_bc2 [1..n]
          where
          f_bc2  a = mappend f_c2 [a .. n]
                     where
                     f_c2  b = mappend f_2 [b .. n]
                               where
                                f_2  c = if (a^2 + b^2 == c^2)
                                         then [(a,b,c)]
                                         else []
```

3. Pattern Matching

The general layout of pattern matching:

fun $p_{1,1}$ ... $p_{1,n}$ = $expr_1$
.
.
.
fun $p_{m,1}$ ... $P_{m,n}$ = $expr_m$

Translation Scheme:

fun a1 ... an => if (*cond*$_1$) then let *defs*$_1$ in $expr_1$
else if (*cond*$_2$) then let *defs*$_2$ in $expr_2$
.
.
.
else if (*cond*$_m$) then let *defs*$_m$ in exprm
else error "fun: no matching pattern"

If pattern contains constructors, type information is needed at run time. We must verify the argument actually matches the constructor specified in the pattern.

- Need two functions

  _type_contr : returns the constructor tag of any type elmt.

  _type_field n : returns the nth field of a structured type.

```
take a1 a2 = if (a1 == 0) then
                 let xs = a2 in []
             else if (a2 == []) then
                 let n = a1 in []
             else if (_type_constr a2 == Cons) then
                 let
                    n = a1
                    x = _type_field 1 a2        --get the head
                    xs = _type_field 2 a2       -- get the tail
                 in  x : take (n-1) xs
             else error "take: no matching pattern"
```

..Front End of the compiler

- Nested function definition
  - The functional core deliberately excludes nested functions. Nested functions are translated into their equivalent non-nested forms.
  - One obvious reason is that the target code (in C), do not support nested functions. There are more convincing arguments for choosing to un-nest nested function rather than implement a compiler supports nested function definitions directly (like Pascal).
  - Modern functional languages uses lambda-lifting to un-nest the nested functions:

```
sv_mul scal vec =
                    let
                      s_mul x = scal * x
                    in
                    map s_mul vec
```

```
s_mul_2 scal x = scal * x

sv_mul scal vec =
                    map (s_mul_2 scal) vec
```

Need to lift s_mul. Define a new function s_mul_2 to replace s_mul.

Back End of the compiler..

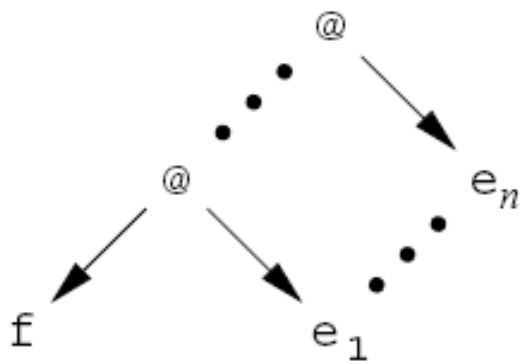
- The task now is to generate C code from the Functional core.
- The generated C code will create data structures, graphs, that are further handled by a graph reducer (an interpreter).
- Graph reducer is the heart of the run-time system, and it deals with higher-order functions, lazy evaluations.

➢ Graph reduction

Notation: write $f\ e_1\ e_2\ \ldots\ e_n$ as $(\ldots((f\ @\ e_1)\ @\ e_2\ )\ldots\ e_n\ )$

where, @ operator denotes an application node holding a function pointer and an argument pointer. The linked list of @ is called an *application spine*.

The graph representation of a function application



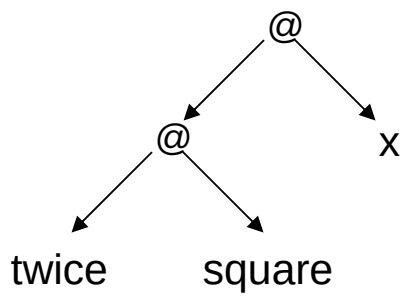
..Back End of the compiler..

How does the graph reducer evaluate an expression?

- The execution of a functional program starts with constructing the graph representing the initial expression.
- Next, the graph reducer repeatedly select a part of the graph (a subexpression) that can be simplified by function application. Such expression is called reducible expression, or redex for short.
- The selected redex is then reduced and this process is repeated.
- The graph reducer stops when it can not find a redex. Finally, the resulting graph is returned as the result of the initial expression.

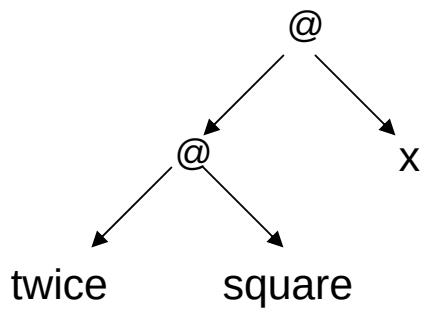
An example of graph reduction:

```
let twice f x = f (f x)
    square n = n*n
in twice square 3
```

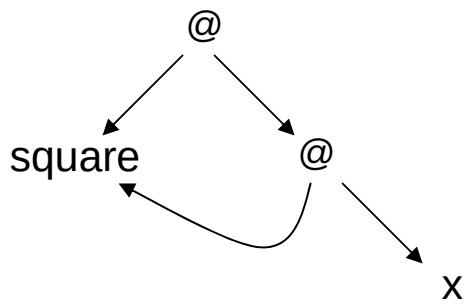


(1)

The initial expression contains one redex, application of twice to its two arguments. From definition of function *twice*, we replace graph (1):

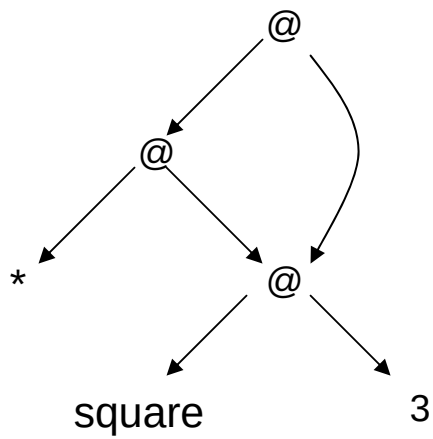


by



(2)

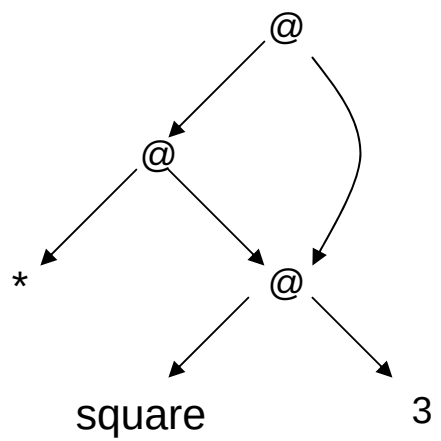
Likewise, from definition of square, we replace graph (2) by (3):



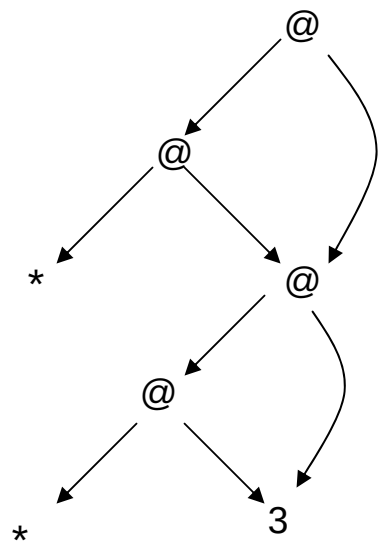
(3)

Continuing the reduction, need to select the next redex in graph (3). There are two candidates: * spine and square spine, each having all their arguments present. The * spine can not be reduced because the built-in operators can only perform operations when all arguments point to a value.

Therefore, graph reducer selects the square spine to reduce from (3) to (4):

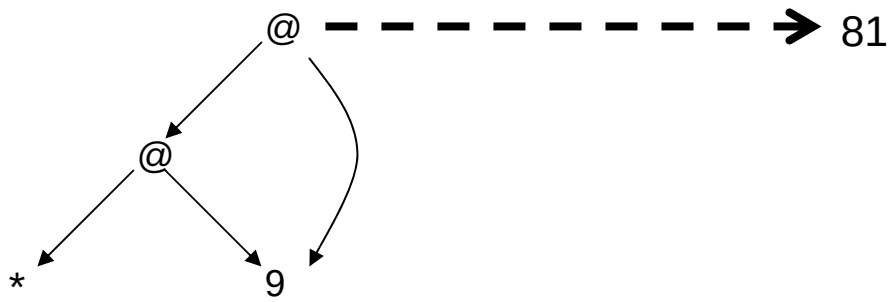


(3)



(4)

- In (4), we observe that the inner * can be reduced, because its arguments are in proper format. Replace (4) by (5), finally the outer * can be reduced to complete the computation:



(5)

The above example shows that the graph reducer is basically a three-stroke engine:

- select a redex
- instantiate the right-hand side of the function
- update the root of the redex

..Back End of the compiler..

- Reduction Order

  A computation graph may contain many redexes. How to select an essential redex to reduce?

  Since the initial expression must reduce to a value, we start with the root of the expression.

  If root is not an application node, the graph reducer returns its value. If the root is an @, we traverse the application spine down to the left to find the function node, say f; we then check whether the application spine contains necessary number of args for f. If it does not, we detect a curried function, and the reducer will return this curried function.

  If all argument are present in the spine for f, and f is a user-defined function, we can apply f. But, in case that f is an built-in operator, it depends on the state of the argument and nature of the operator.

  For example, * and + require the arguments pointing to numbers. We can say the arg of * and + are strict arguments (* and + are strict on all of their arguments).

..Back End of the compiler..

- The process of reducing the graph and finding the next redex is called unwinding.

- The unwinding we see so far always selects the leftmost-outermost redex; this reduction order is known as Normal-order reduction.
  This is the underlying mechanism to implement lazy evaluation.

- Another reduction order, Applicative-order reduction, selects the leftmost-innermost redex; this is the strategy employed by imperative languages (arg evaluated before invoking function).

..Back End of the compiler..

- Implementation of graph reducer

  (the reduction engine in C)

  - Reduction engine operates on 4 types of graph nodes:
    number, list, function application, function descriptor
  - Structure of graph node: a Tag, and a union holding one of the four node types.
  - A function descriptor contains: arity of function, function name, and a pointer to its code (in C).
  - The constructor functions specify how each type of graph node can be constructed.

..Back End of the compiler..

```
typedef enum {FUNC, NUM, NIL, CONS, APPL} node_type;

typedef struct node *Pnode;

typedef Pnode (*unary)(Pnode *arg);

struct function_descriptor {
    int arity;
    const char *name;
    unary code;
};

struct node {
    node_type tag;
    union {
        struct function_descriptor func;
        int num;
        struct {Pnode hd; Pnode tl;} cons;
        struct {Pnode fun; Pnode arg;} appl;
    } nd;
};

/* Constructor functions */
extern Pnode Func(int arity, const char *name, unary code);
extern Pnode Num(int num);
extern Pnode Nil(void);
extern Pnode Cons(Pnode hd, Pnode tl);
extern Pnode Appl(Pnode fun, Pnode arg);
```

<Declaration of node types and constructor functions>

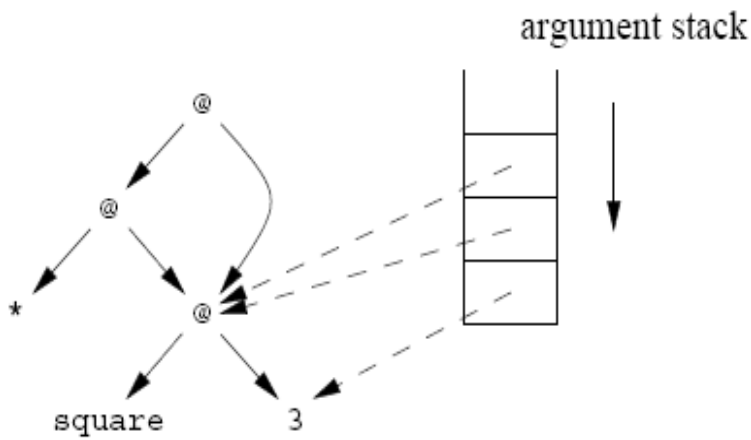
..Back End of the compiler..

```
#define STACK_DEPTH     10000

static Pnode arg[STACK_DEPTH];
static int top = STACK_DEPTH;                    /* grows down */

Pnode eval(Pnode root) {
    Pnode node = root;
    int frame, arity;

    frame = top;
    for (;;) {
        switch (node->tag) {
        case APPL:                                       /* unwind */
            arg[--top] = node->nd.appl.arg;  /* stack argument */
            node = node->nd.appl.fun;       /* application node */
            break;

        case FUNC:
            arity = node->nd.func.arity;
            if (frame-top < arity) {        /* curried function */
                top = frame;
                return root;
            }
            node = node->nd.func.code(&arg[top]);     /* reduce */
            top += arity;                    /* unstack arguments */
            *root = *node;                /* update root pointer */
            break;

        default:
            return node;
        }
    }
}
```

..Back End of the compiler..

- The eval function implements the reduction engine. It takes a pointer to an graph node and perform case analysis on Tags to find out what to do next.
- Application nodes and function descriptor require further processing, while other tags will stop the reduction engine because basic value can be returned immediately.
- To process a @, we must unwind it to obtain the function and its arguments. Arguments are pushed onto stack so they can be referred directly when needed.

  For example:



..Back End of the compiler..

- When the reduction engine detects a function descriptor, it checks if all arguments are present. If this is not the case, a curried function is detected, reduction stops, and arguments are unstacked before returning.
- If, however, all arguments are present, the pointer to the function code is retrieved from descriptor and function is called. When function returns, the args are popped off the stack, and the result is copied to the root node (of the function application).
- eval is called recursively.

- Reduction engine is accompanied by routines implementing built-in operators and list operators. Each built-in operator is wrapped in a function descriptor, which includes a pointer to the code performing the reduction.

..Back End of the compiler..

Implementation of built-in opr *

```
Pnode mult (Pnode _arg0, Pnode _arg1)
{
Pnode a = eval(_arg0);
Pnode b = eval(_arg1);
return Num(a->nd.num * b->nd.num);
}
Pnode _mult(Pnode *arg)
{
return mult(arg[0], arg[1]);
}

struct node __mult= {FUNC, {2, "mul", _mult}};
```

__mult represents function *mult* as a graph node; __mult may be used by code generator to construct graph for expression like *(a,b).

..Back End of the compiler..

- Code generation for functional core

  - Note: this is not ordinary program transformation between these two.

  - The (only) task of the generated C code is to create graph for the graph reducer to reduce. And that graph represents the source program.

  - For each function definition, we generate a function descriptor and code to build the right-hand-side.

..Back End of the compiler..

Example: twice f x = f (f x)

```
Pnode twice (Pnode _arg0, Pnode _arg1)
{
Pnode f  = eval(_arg0);
Pnode x = eval(_arg1);
return Appl(f, (Appl(f, x) ) );
}

Pnode _twice(Pnode *arg)
{
return twice(arg[0], arg[1]);
}

struct node __twice=
        {FUNC, {2, “twice", _twice} }
```

Generating code for functions that instantiate the right-hand side of the function at run-time is straightforward.

..Back End of the compiler..

- Generating code for other core constructs:
  - The graph reducer knows how to handle function applications, so we can simply translate each constructs into a function call.

  - Identifier : we can use the same name in C code, therefore, need no further processing.
  - Numbers : wrap them with the Num constructor function (eg for 5, Num(5))
  - Function application: build a graph consisting application nodes, which hold pointers to argument expression.
  - If-then-else: less straightforward. Need to use conditional expression in C, (cond ? x : y).

..Back End of the compiler..

- Let-expression: the let bindings are translated to assignment to local variables in C. Such local variable holds pointer to the nodes representing named expression.

  Example

```
f a = let
        b = fac a
        c = b * b
      in
        c + c
```

```
Pnode f(Pnode _arg0)
{
   Pnode a = _arg0;
   Pnode b = Appl(&__fac, a);
   Pnode c = Appl(Appl(&__mul, b), b);
   return Appl (Appl(&__add, c), c);
}
```

..Back End of the compiler..

- Optimizing the functional core

  - Functional language compilers usually apply a number of advanced optimization techniques to achieve an acceptable performance.

  - Typically, these optimizations are performed on the AST (representing intermediate code).

  - Introduce one optimization technique, Strictness analysis, tackling the overhead of manipulating graph node.

- Strict analysis
  - Lazy evaluation causes expression at argument positions of a function call to be passed unevaluated as a graph node.
  - However, building graph is very expensive, and becomes overhead when the function will eventually evaluate the (lazy) arguments to proceed it computation.
  - In fact, some functions always need to evaluate their args.
  - If a function always needs the value of its argument a_i, the function is said to be strict on a_i.

  - Purpose of strictness analysis:

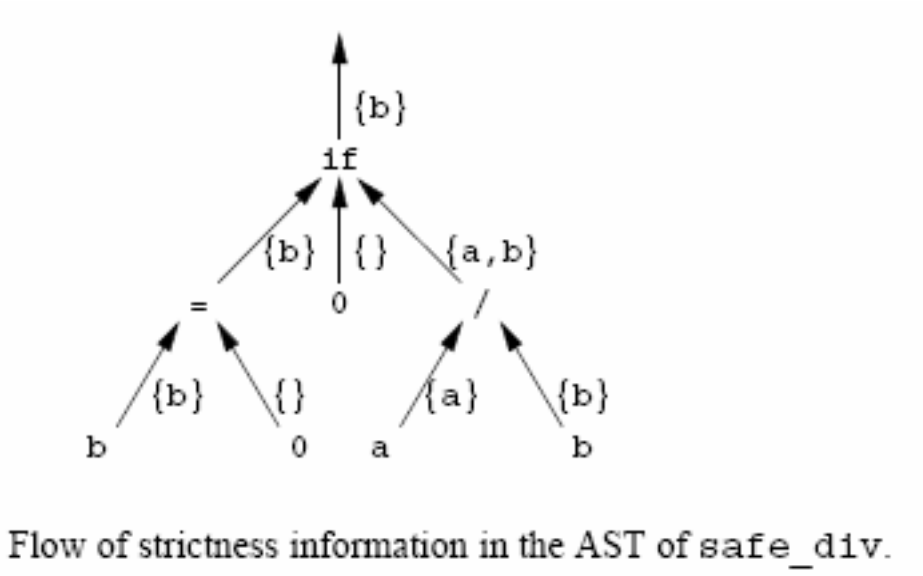
    Determine all functions which have strict arguments?. Then the code generator can evaluating expression at strict arg positions before invoking function. Thus, we can save the overhead of:
    1. building graph by the caller ;
    2. call to the graph reducer by the callee.

..Back End of the compiler..

- Analyzing the strictness of a function
  - Propagate the strictness of built-in operator and other functions bottom-up through the AST.

    Example

safe_div a b = if (b==0) then 0 else a/b



Flow of strictness information in the AST of `safe_div`.

The derived set of strict argument for safe_div is {b}

- Strictness propagation rules

• We start from identifiers at the leaves position of the AST. A parameter p will yield the set {p}.

• Likewise, local variable v yield the set {v}.

• Constant, function names yield empty set { }.

• Then applying propagation rules, as follows:

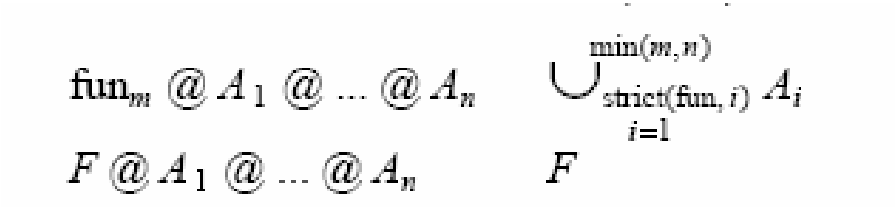
Language construct	Set to be propagated	
L operator R	$L \cup R$	
`if` C `then` T `else` E	$C \cup (T \cap E)$	
$\text{fun}_m \ @ \ A_1 \ @ \ \ldots \ @ \ A_n$	$\bigcup_{i=1}^{\min(m,n)}{}_{\text{strict(fun, } i)} A_i$	
$F \ @ \ A_1 \ @ \ \ldots \ @ \ A_n$	F	
`let` v = V `in` E	$(E \setminus \{v\}) \cup V$	, if v $\in E$
	E	, otherwise

Capital letters on the left column denotes language construct, whereas in the right column they denotes the corresponding set of strict variables.

Predicate strict(fun,i) = true if arg_i of fun is strict

false otherwise

- Note that there are two rules for function applications:



The first rule covers the case that the function identifier is a global name, fun_m indicates its arity is m. In this case, all args set occurring at strict positions are propagated.

The second rule covers the cases that function is described by some expression(F). For example, F may be a result from computing a higher-order function. So no strictness information about the args can be derived.

- Strict analysis for recursive function definitions

The solution is to optimistically assume that a recursive function is strict on all its arguments, and then check if the derived set (R) includes all arguments (set A). In general case, R do not contain all arguments, so we know the assumption (set A) was too optimistic. Let A=R, and derive the strict arg set (call it R') again, check if R'=A. Continue with this narrowing operation, until R'=A.

..Back End of the compiler

- Code generation for strict arguments
  - Expression occurring at strict arg positions may be evaluated immediately without building graph.
  - Consider factorial function fac

    fac = if (n==0) then 1 else n*fac(n-1)

```
Pnode fac(Pnode _arg0)
{
   Pnode n = _arg0;
   return equal(n, Num(0))- >nd.num ? Num(1) :
   mul(n, fac(Appl(Appl(&__sub, n), Num(1))));
}
```

Without strictness information, the arg of fac is passed lazily as a graph node.

```
Pnode fac(Pnode _arg0)
{
  Pnode n = _arg0;
  return equal(n, Num(0))->nd.num ? Num(1) :
  mul(n, fac(sub(n, Num(1))));
}
```

Using the fact that fac is strict in its arg, we can remove the application spine.

The End..

Thanks.