Compiling Haskell to LLVM

John van Schie

Center for Software Technology, Universiteit Utrecht
http://www.cs.uu.nl/groups/ST

June 26, 2008

Daily supervisors:
dr. A. Dijkstra
drs. J.D. Fokker

Second supervisor:
prof. dr. S.D. Swierstra

Overview

Typical compiler pipeline

1. Parse language to an abstract syntax tree (AST)
2. Transform the AST (optimization, simplification, etc.)
3. Generate executable

Typical compiler pipeline

1. Parse language to an abstract syntax tree (AST)
2. Transform the AST (optimization, simplification, etc.)
3. Generate executable

In this talk, we focus on the backend of the compiler.

Possible targets for generating executables

- Native assembly
  - Very fast

x86 assembly

```
f:
pushl   %ebp
movl    %esp, %ebp
subl    $8, %esp
movl    12(%ebp), %eax
movl    %eax, (%esp)
call    g
imull   8(%ebp), %eax
leave
ret
```

Possible targets for generating executables

- Native assembly
    - Very fast
- High level languages (C, Java, etc.)
    - Portable
    - Fast

C

```
int f( int x, int y )
{
  return x * g( y );
}
```

Possible targets for generating executables

- Native assembly
  - Very fast
- High level languages (C, Java, etc.)
  - Portable
  - Fast
- Managed virtual environments (JVM, CLI, etc)
  - Portable
  - Rich environment

Java byte code

```
static int f(int, int);
  Code:
   0:iload_0
   1:iload_1
   2:invokestatic #2; //g
   5:imul
   6:ireturn
```

Possible targets for generating executables

- Native assembly
  - Very fast
- High level languages (C, Java, etc.)
  - Portable
  - Fast
- Managed virtual environments (JVM, CLI, etc)
  - Portable
  - Rich environment
- Typed assembly languages
  - What are they?

LLVM assembly

```
define i32 @f( i32 %x
             , i32 %y )
{
  %vr.0 = call i32
          @g( i32 %y )
  %vr.1 = mul i32 %x, %vr.0
  ret i32 %vr.1
}
```

Typed assembly languages

Definition

- Architecture neutral
- Statically typed
- Optionally high level features

Typed assembly languages

Definition

- Architecture neutral
- Statically typed
- Optionally high level features

Goal

A universal intermediate representation that high level languages can be mapped to.

Typed assembly languages

Definition

- Architecture neutral
- Statically typed
- Optionally high level features

Goal

A universal intermediate representation that high level languages can be mapped to.

Advantages:

- Portable
- Very fast
- Very flexible

Possible targets for Haskell compilers

- Native assembly:
  - Hard to maintain
- C:
  - No efficient tail call support
- Managed virtual environments:
  - Run-time system optimized for imperative languages

Possible targets for Haskell compilers

- Native assembly:
  - Hard to maintain
- C:
  - No efficient tail call support
- Managed virtual environments:
  - Run-time system optimized for imperative languages

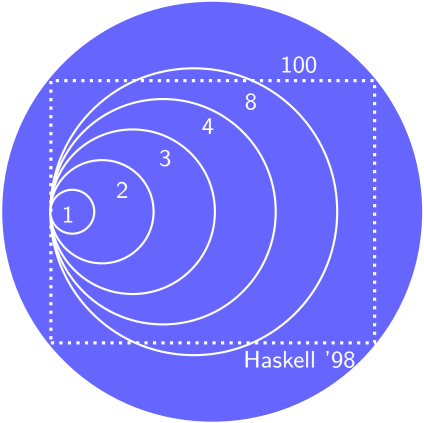
Are typed assembly languages suitable targets for Haskell compilers?

EHC

The Essential Haskell Compiler (EHC).

- Actively developed at Utrecht University by Atze Dijkstra and Jeroen Fokker
- Executable generation via C
- Allows for easy experimentation with 'variants'

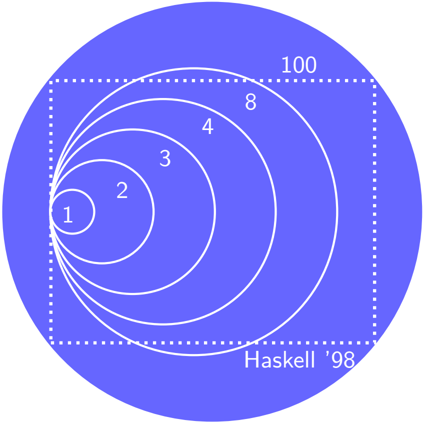
EHC variants



EHC is a sequence of compilers

- Variant 1: Explicitly typed lambda calculus
- Variant 8: Code generation
- Variant 100: Full Haskell 98, some extensions

EHC variants



EHC is a sequence of compilers

- Variant 1: Explicitly typed lambda calculus
- Variant 8: Code generation
- Variant 100: Full Haskell 98, some extensions

We will use variant 8 for all examples in this talk.

LLVM

The Low Level Virtual Machine (LLVM) compiler infrastructure project.

- Actively developed at Apple, main developer Chris Lattner
- Mature optimizing compiler framework
- Supports 11 different architectures, including x86, PowerPC, Alpha, and SPARC

The LLVM assembly language

- Infinite amount of virtual registers
- RISC format
- Stack management
- Strongly typed
- Static single assignment (SSA)

The LLVM assembly language - Example

LLVM assembly example - increment counter

```
@globalCounter = global i32 0

define fastcc void @incrGlobalCounter( i32 %by )
{
  %vr0 = load i32* @globalCounter
  %vr1 = add i32 %vr0 , %by
   store i32 %vr1 , i32* @globalCounter
   ret void
}
```

The LLVM assembly language - Example

%vr*n* are virtual registers

LLVM assembly example - increment counter

```
@globalCounter = global i32 0

define fastcc void @incrGlobalCounter( i32 %by )
{
  %vr0 = load i32* @globalCounter
  %vr1 = add i32 %vr0, %by
  store i32 %vr1, i32* @globalCounter
  ret void
}
```

The LLVM assembly language - Example

%vr*n* are virtual registers

@globalCounter refers to memory

LLVM assembly example - increment counter

```
@globalCounter = global i32 0

define fastcc void @incrGlobalCounter( i32 %by )
{
  %vr0 = load i32* @globalCounter
  %vr1 = add i32 %vr0, %by
  store i32 %vr1, i32* @globalCounter
  ret void
}
```

The LLVM assembly language - Example

%vr*n* are virtual registers

@globalCounter refers to memory

High level function calls

LLVM assembly example - increment counte

```
@globalCounter = global i32 0

define fastcc void @incrGlobalCounter( i
{
  %vr0 = load i32* @globalCounter
  %vr1 = add i32 %vr0 , %by
   store i32 %vr1 , i32* @globalCounter
   ret void
}
```

The LLVM assembly language - Example

%vr*n* are virtual registers

@globalCounter refers to memory

High level function calls

All operations are typed.

LLVM assembly example - increment counte

```
@globalCounter = global i32 0

define fastcc void @incrGlobalCounter( i
{
  %vr0 = load i32* @globalCounter
  %vr1 = add i32 %vr0, %by
  store i32 %vr1, i32* @globalCounter
  ret void
}
```

The LLVM assembly language - Example

%vr*n* are virtual registers

@globalCounter refers to memory

High level function calls

All operations are typed.

Each virtual register assigned once.

LLVM assembly example - increment counte

```
@globalCounter = global i32 0

define fastcc void @incrGlobalCounter( i
{
  %vr0 = load i32* @globalCounter
  %vr1 = add i32 %vr0 , %by
  store i32 %vr1 , i32* @globalCounter
  ret void
}
```

Research questions

Question: Is the LLVM assembly language a suitable target for EHC?

1. *a)*: Is the implementation as easy as targeting C?
2. *b)*: Is the generated code efficient?

Research questions

Question: Is the LLVM assembly language a suitable target for EHC?

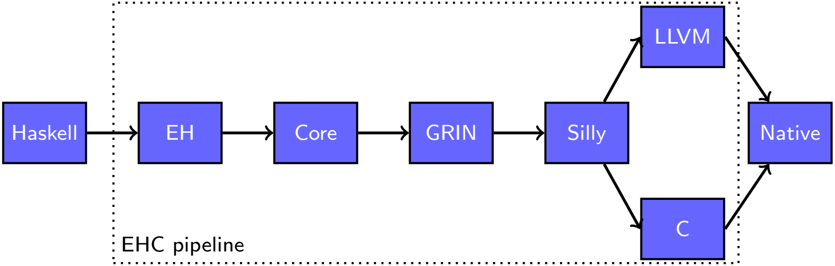
1. *a)*: Is the implementation as easy as targeting C?
2. *b)*: Is the generated code efficient?

Contributions

- An implementation of a EHC backend that creates executables via LLVM assembly.
- A comparison of execution time and memory usage between the LLVM and C targeting backends.
- Suggestions for more efficient code generation by EHC and the impact of these changes.

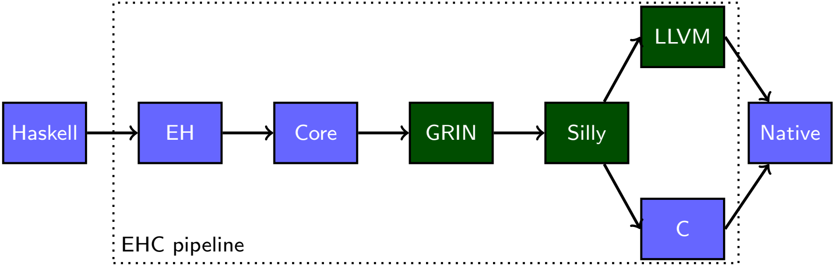
Implementation

Pipeline stages



- *Essential Haskell (EH)*: desugared Haskell
- *Core*: lambda calculus with some extensions
- *GRIN*: makes evaluation order of the program explicit
- *Silly*: foundation for translation to imperative languages

Pipeline stages



- *Essential Haskell (EH)*: desugared Haskell
- *Core*: lambda calculus with some extensions
- *GRIN*: makes evaluation order of the program explicit
- *Silly*: foundation for translation to imperative languages

Running example

Running example

```haskell
fib :: Int -> Int
fib n    | n == 0   = 0
         | n == 1   = 1
         | True     = fib (n-1) + fib (n-2)

main = fib 33
```

GRIN - Overview

The Graph Reduction Intermediate Notation (GRIN)

- Developed by Urban Boquist
- Makes expression evaluation order explicit
- A very efficient evaluation model
- Closed world assumption

GRIN - Representation of expressions

- Expressions represented as nodes.

Definition of nodes

A node is a sequence of fields, where the first field is a tag, followed by zero or more payload fields.

- 3 type of nodes:
  - Constructed values (C)
  - Suspended functions (F)
  - Partial applications (P)

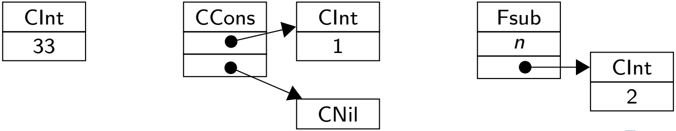
GRIN - Representation of expressions

- Expressions represented as nodes.

Definition of nodes

A node is a sequence of fields, where the first field is a tag, followed by zero or more payload fields.

- 3 type of nodes:
  - Constructed values (C)
  - Suspended functions (F)
  - Partial applications (P)



GRIN - Haskell to GRIN

- Erase type information
- Transform to SSA form
- Explicit access to memory
- Add evaluation order to program
  - Expressions translated to a graph
  - Graph evaluated to WHNF

GRIN - Generated code

Recursive case of fib

```
$fib $p1 =
  [...]
  store (CInt 2); λ$p2 →
  store (Fsub $p1 $p2); λ$p3 →
  store (Ffib $p3); λ$p4 →
  store (CInt 1); λ$p5 →
  store (Fsub $p1 $p5); λ$p6 →
  store (Ffib $p6); λ$p7 →
  store (Fadd $p7 $p4); λ$p8 →
  $eval $p8
```

GRIN - Generated code

Each variable is assigned exactly once.

Recursive case of fib

```
$fib $p1 =
  [...]
  store (CInt 2); λ$p2 →
  store (Fsub $p1 $p2); λ$p3 →
  store (Ffib $p3); λ$p4 →
  store (CInt 1); λ$p5 →
  store (Fsub $p1 $p5); λ$p6 →
  store (Ffib $p6); λ$p7 →
  store (Fadd $p7 $p4); λ$p8 →
  $eval $p8
```

GRIN - Generated code

Recursive case of fib

```
$fib $p1 =
  [...]
  store (CInt 2); λ$p2 →
  store (Fsub $p1 $p2); λ$p3 →
  store (Ffib $p3); λ$p4 →
  store (CInt 1); λ$p5 →
  store (Fsub $p1 $p5); λ$p6 →
  store (Ffib $p6); λ$p7 →
  store (Fadd $p7 $p4); λ$p8 →
  $eval $p8
```

Each variable is assigned exactly once.

Only *store*, *update*, and *load* perform memory access.

GRIN - Generated code

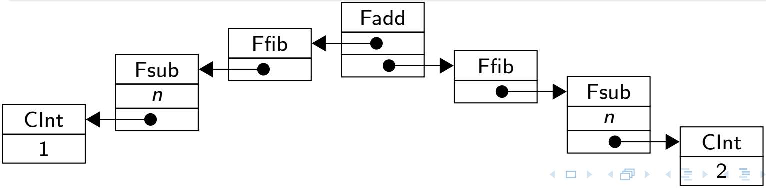
Recursive case of fib

```
$fib $p1 =
  [...]
  store (CInt 2); λ$p2 →
  store (Fsub $p1 $p2); λ$p3 →
  store (Ffib $p3); λ$p4 →
  store (CInt 1); λ$p5 →
  store (Fsub $p1 $p5); λ$p6 →
  store (Ffib $p6); λ$p7 →
  store (Fadd $p7 $p4); λ$p8 →
  $eval $p8
```

Each variable is assigned exactly once.

Only *store*, *update*, and *load* perform memory access.

Graph is build for $fib(n-1)+fib(n-2)$



GRIN - Generated code

Recursive case of fib

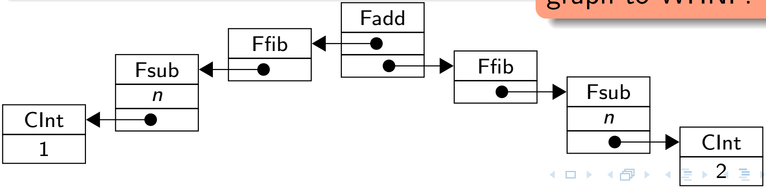
```
$fib $p1 =
  [...]
  store (CInt 2); λ$p2 →
  store (Fsub $p1 $p2); λ$p3 →
  store (Ffib $p3); λ$p4 →
  store (CInt 1); λ$p5 →
  store (Fsub $p1 $p5); λ$p6 →
  store (Ffib $p6); λ$p7 →
  store (Fadd $p7 $p4); λ$p8 →
  $eval $p8
```

Each variable is assigned exactly once.

Only *store*, *update*, and *load* perform memory access.

Graph is build for $fib(n-1)+fib(n-2)$

eval evaluates the graph to WHNF.



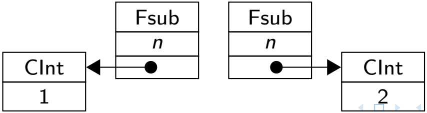
Silly - GRIN to Silly

- Abstracts over the transformations of GRIN to an imperative language
- Language has an imperative feel
- Consists of only well supported constructs (assignment, switch, function call, variable, constant etc.)
- Decides on physical representation of nodes
  - A node is an array of *heap cells*
- Introduces local and global variables
  - Global variables: return node and constant applicable form nodes
  - Local variables: value depends on control flow

Silly - Generated code

Recursive case of fib

```
p3     := allocate(3) { GCManaged };
p3[0]  := Fsub;
p3[1]  := p1;
p3[2]  := global_p2;
p6     := allocate(3) { GCManaged };
p6[0]  := Fsub;
p6[1]  := p1;
p6[2]  := global_p5;
fun_fib(p6);
i72    := RP[1];
fun_fib(p3);
i92    := foreign primAddInt(i72, RP[1]);
```

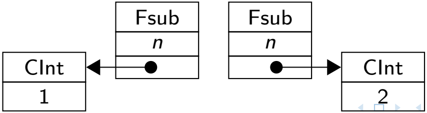


Silly - Generated code

Array indexes used to access fields of nodes

Recursive case of fib

```
p3     := allocate(3) { GCManaged };
p3[0]  := Fsub;
p3[1]  := p1;
p3[2]  := global_p2;
p6     := allocate(3) { GCManaged };
p6[0]  := Fsub;
p6[1]  := p1;
p6[2]  := global_p5;
fun_fib(p6);
i72    := RP[1];
fun_fib(p3);
i92    := foreign primAddInt(i72, RP[1]);
```



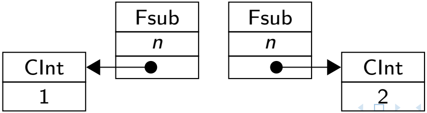
Silly - Generated code

Array indexes used to access fields of nodes

*i*72, *i*92 are local variables.

Recursive case of fib

```
p3     := allocate(3) { GCManaged };
p3[0]  := Fsub;
p3[1]  := p1;
p3[2]  := global_p2;
p6     := allocate(3) { GCManaged };
p6[0]  := Fsub;
p6[1]  := p1;
p6[2]  := global_p5;
fun_fib(p6);
i72    := RP[1];
fun_fib(p3);
i92    := foreign primAddInt(i72, RP[1]);
```



Silly - Generated code

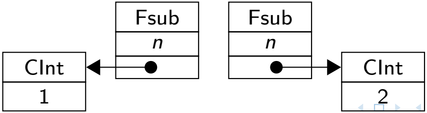
Recursive case of fib

```
p3     := allocate(3) { GCManaged };
p3[0]  := Fsub;
p3[1]  := p1;
p3[2]  := global_p2;
p6     := allocate(3) { GCManaged };
p6[0]  := Fsub;
p6[1]  := p1;
p6[2]  := global_p5;
fun_fib(p6);
i72    := RP[1];
fun_fib(p3);
i92    := foreign primAddInt(i72, RP[1]);
```

Array indexes used to access fields of nodes

*i*72, *i*92 are local variables.

Prefix *global_* for global variables.



Silly - Generated code

Recursive case of fib

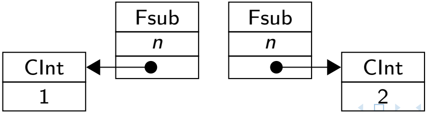
```
p3      := allocate(3) { GCManaged };
p3[0]   := Fsub;
p3[1]   := p1;
p3[2]   := global_p2;
p6      := allocate(3) { GCManaged };
p6[0]   := Fsub;
p6[1]   := p1;
p6[2]   := global_p5;
fun_fib(p6);
i72     := RP[1];
fun_fib(p3);
i92     := foreign primAddInt(i72, RP[1]);
```

Array indexes used to access fields of nodes

*i*72, *i*92 are local variables.

Prefix *global_* for global variables.

RP is the global return array.



Generating LLVM assembly

LLVM - Silly to LLVM

- Infer LLVM types
- Generate LLVM instructions
- Extract C strings

LLVM - Inferring LLVM types

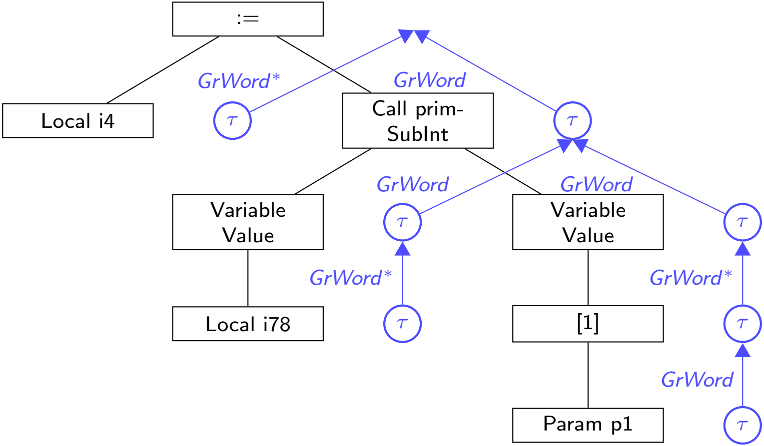
- Types are erased in early stages of the pipeline
- To allow re-typing, we use the following trick:
  - Define *GrWord* as an integer with the same size as a native pointer
  - This allows us to store integers and pointers at the same location
  - Example: CCons node
- Each Silly statement is typed in isolation.
- Instruction generation uses type information to insert type conversions.

LLVM - Inferring LLVM types (2)

- We have the following assumptions:
    - Global variables: *GrWord***
    - Local variables: *GrWord**
    - Parameter variables: *GrWord*
    - Allocations: *GrWord*
    - Constants: *GrWord*
- Each leaf in a Silly AST matches one of these cases.
- Other nodes use types of children and local information.
- Easy to implement with an attribute grammar.

LLVM - Inferring LLVM types (3)

Type inference of *i4 := foreign primSubInt(i78, p1[1]);*



LLVM - Generate instructions

- Instruction generation is a 3 step process:
  - Acquire result variables from child AST nodes.
  - Generate code to convert the types of result variables to the expected types.
  - Generate code for the semantics of this node.

LLVM - Generate instructions (2)

i4 := foreign primSubInt(i78, p1[1]);

```
; Convert p1 to GrWord*
%vr0 = inttoptr i32 %p1 to i32*

; Get pointer to field 1 of p1
%vr1 = getelementptr i32* %vr0, i32 1

; Load pointer to field 1 of p1
%vr2 = load i32* %vr1

; Load local variable i78 from memory
%vr3 = load i32* %i78

; Do function call
%vr4 = call i32 @primSubInt( i32 %vr3, i32 %vr2 )

; Store result in i4
store i32 %vr4, i32* %i4
```

LLVM - Extracting C Strings

- Strings not part of executable code.
- Inline strings collected and defined constant.
- Implemented with 1 AG threaded attribute.

LLVM - Implementation compared to C

- Although LLVM is more low level, the backend implementation does not differ much:
  - GRIN prepares for SSA form.
  - Silly abstracts over imperative backends.
  - All Silly constructs easily mappable to LLVM.
  - Attribute grammars help keep implementation of bottom-up algorithms concise.

Results

Results - Benchmark situation

Comparing the C and LLVM backend by compiling 8 nofib programs.

We measure the pure gain of targeting LLVM instead of C[a].

- Both compiled with full optimization.
- Both use the same run time system.
- Both allocate with `malloc()` and do not de-allocate.
- Machine had enough memory to avoid swapping.

[a] compiled with GCC

Results - The numbers

program	time (Δ%)	memory allocated (Δ%)
digits-of-e1	24.1%	
digits-of-e2	8.1%	
exp3_8	-9.4%	
primes	8.8%	
queens	17.7%	
tak	23.6%	
wheel-sieve1	22.1%	
wheel-sieve2	10.2%	
average	13.1%	

Positive Δ: LLVM backend performs better than the C backend.

Results - The numbers

program	time (Δ%)	memory allocated (Δ%)
digits-of-e1	24.1%	0.1 %
digits-of-e2	8.1%	0.2 %
exp3_8	-9.4%	0.6 %
primes	8.8%	1.1 %
queens	17.7%	0.6 %
tak	23.6%	1.4 %
wheel-sieve1	22.1%	0.1 %
wheel-sieve2	10.2%	2.0 %
average	13.1%	0.8 %

Positive Δ: LLVM backend performs better than the C backend.

Results - The numbers explained

Why is the LLVM compiler more efficient?

- C backend uses shadow stack for tail calls
- C used as portable assembly
- Aliasing problem

Results - The numbers explained

Why is the LLVM compiler more efficient?

- C backend uses shadow stack for tail calls
- C used as portable assembly
- Aliasing problem

Slowdown of the program exp3_8:

- Native backend of the C compiler does a better job.
- This is work in progress for LLVM.

Conclusion

Future work

- Simplify the implementation of the backend.
  - Perform typing in Silly
  - (alternative) Propagate types to the back end.

Future work

- Simplify the implementation of the backend.
  - Perform typing in Silly
  - (alternative) Propagate types to the back end.
- Increase performance of the generated code:
  - Return result of functions in registers instead of a global variable
  - Allocate global variables statically
  - More efficient memory management

Future work

- Simplify the implementation of the backend.
  - Perform typing in Silly
  - (alternative) Propagate types to the back end.
- Increase performance of the generated code:
  - Return result of functions in registers instead of a global variable
  - Allocate global variables statically
  - More efficient memory management
- Validate suitability of LLVM as backend target with other Haskell compilers

Conclusion

For EHC, LLVM is a more favourable target than C.

- Complexity comparable to generation of C
- Run time reduced with 13.1% by average

There is still much room for improvement, not a production compiler yet.