Declarative Language Definitions and Code Generation as Linearization

TYPES Meeting, Jouy-en-Josas, December 15–18, 2004.

Aarne Ranta

aarne@cs.chalmers.se

The Challenge

3. Write a C compiler. The input language is a subset of C. A valid input program should produce byte code than when executed runs the C program.

Lennart Augustsson, programming challenges posted to the participants of the Dagstuhl workshop on Dependent Types in Programming, 2004.

Example program: C source

```
int factr (int n) {
  int f ;
  if (n < 2) {
    f = 1 ;
    }
  else {
    f = n * factr (n-1) ;
  }
  return f ;
} ;
int main () {
  int n = 1 ;
  while (n < 11) {
      printf("%d",factr(n)) ;
      n = n+1 ;
    }
  return ;
} ;
```

Compiling the program

```
[aarne@localhost compiler]$ gfcc factr.c
> > wrote file factr.j
Generated: factr.class
```

Generated JVM bytecode (Jasmin assembler format)

```
.class public factr
.super java/lang/Object
.method public <init>()V
aload_0
invokenonvirtual java/lang/Object/<init>()V
return
.end method
.method public static factr(I)I
.limit locals 3
.limit stack 1000 ;
; alloc i n i ;
; alloc i f ;
iload 0
ldc 2 ;
invokestatic runtime/ilt(II)I
ifeq FALSE_
ldc 1 ;
istore 1
```

```
goto TRUE_
FALSE_:
iload 0
iload 0
ldc 1 ;
isub
invokestatic factr/factr(I)I
imul
istore 1
TRUE_:
iload 1
ireturn
.end method ;

.method public static main([Ljava/lang/String;)V
.limit locals 2
.limit stack 1000 ;
; alloc i n ;
ldc 1 ;
istore 1
```

```
TEST_:
iload 1
ldc 11 ;
invokestatic runtime/ilt(II)I
ifeq END_
iload 1
invokestatic factr/factr(I)I
invokestatic runtime/iprintf(I)V
iload 1
ldc 1 ;
iadd
istore 1
goto TEST_
END_:
return ;
.end method ;
```

Running the bytecode

```
[aarne@localhost compiler]$ java factr
1
2
6
24
120
720
5040
40320
362880
3628800
```

Compiler Passes

```
x + 5
```

Lexer

```
[TV "x",  TS "+", TI "5"]
```

Parser

```
EAdd (EVar "x") (ENum 5)
```

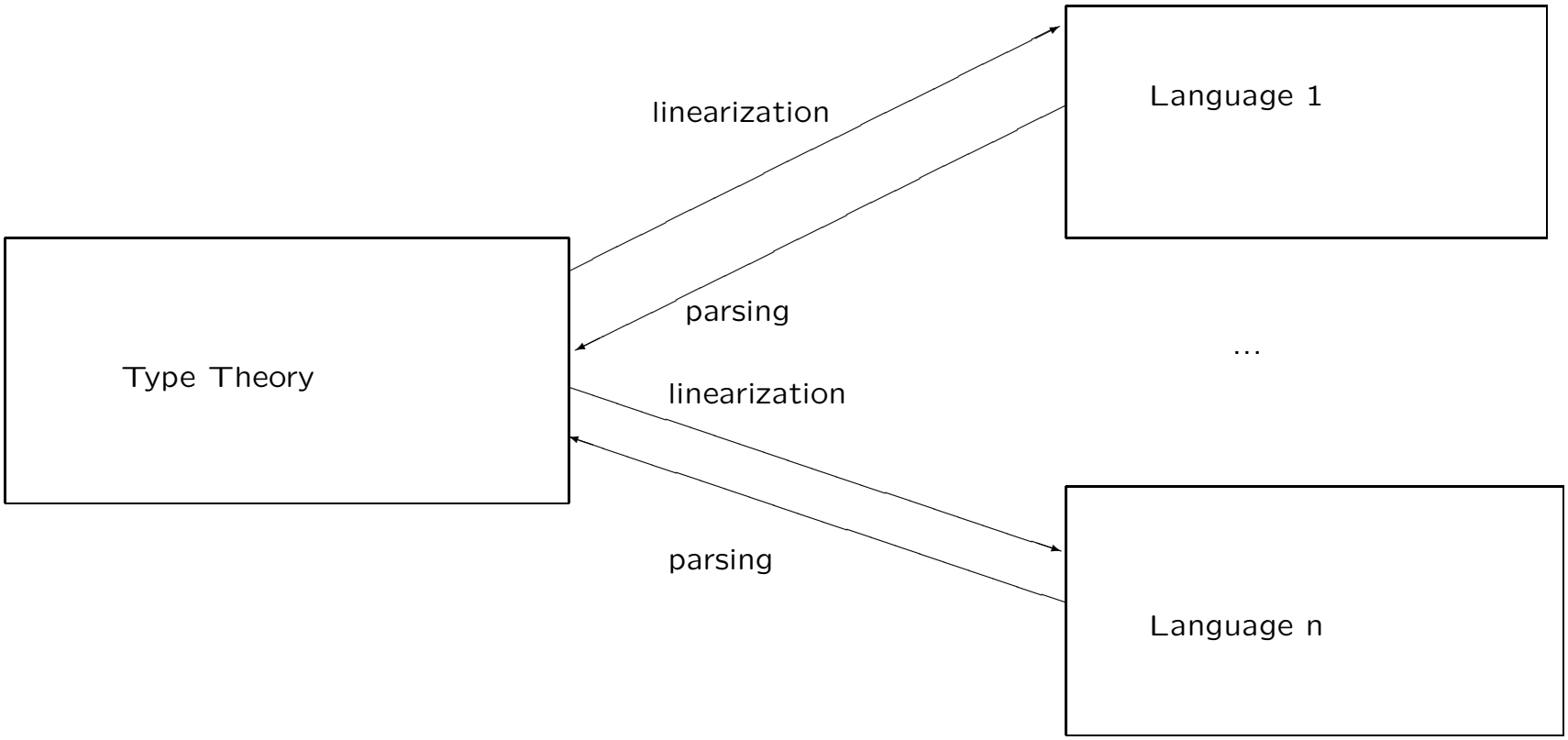
Type annotator

```
EAddT TInt (EVarT TInt "x") (ENum TInt 5)
```

Code generator

```
iload_1 ; bipush 5 ; iadd
```

GF = Grammatical Framework



Example GF rules

Abstract syntax:

```
fun Ev : Nat -> Prop ;
```

Concrete syntax of English:

```
lin Ev x = {s = x.s ++ "is" ++ "even"} ;
```

Concrete syntax of French:

```
lin Ev x = {
  s = table {m => x.s ++
        table {Ind  => "est" ;  Subj => "soit"} ! m ++
        table {Masc => "pair" ; Fem  => "paire"} ! x.g
    }
  } ;
```

Abstract and concrete syntax

Abstract Syntax is defined in a Logical Framework:

```
cat C (x1 : A1)...(xn : An)
fun f : A
```

Concrete syntax is defined by linearization:

```
lincat C = T
lin    f = t
```

Notice: linearization type T instead of just strings.

Linearization

Well-typedness:

$$\frac{f\colon A_1 \;\to\; \cdots \;\to\; A_n \;\to\; A}{f^{\circ}\colon A_1^{\circ} \;\to\; \cdots \;\to\; A_n^{\circ} \;\to\; A^{\circ}}$$

Linearization algorithm:

$$(f\ a_1\ \ldots\ a_n)^{\circ} \;=\; f^{\circ}\ a_1^{\circ}\ \ldots\ a_n^{\circ}$$

Expressions: abstract syntax

Expressions of a type; the "class" of numeric types

```
cat
  Typ ;
  Exp Typ ;
  IsNum Typ ;
fun
  EAdd : (n : Typ) -> IsNum n -> Exp n -> Exp n -> Exp n ;

  TInt : NumTyp ;
  isNumInt : IsNumTyp TInt ;
```

The parsing and type annotation phases

```
x + 5
```

Lexer + parser

```
EAdd ? ? (EVar ? x) (EInt 5)
```

Type checker + metavariable solver

```
EAdd TInt isNumInt (EVar TInt x) (EInt 5)
```

Expressions: concrete syntax of C

Type: record with string and precedence

```
lincat Exp = {s : Str ; p : Prec} ;

oper   Prec : PType = Predef.Ints 4 ; -- = {0,1,2,3,4}
```

Example:

```
lin EInt n = {s = n.s ; p = 4} ;
```

Auxiliary operations for precedence

```
PrecExp : Type = {s : Str ; p : Prec} ;

usePrec : PrecExp -> Prec -> Str = \x,p ->
  ifThenStr (less x.p p)
    (paren x.s)
    x.s ;

infixL : Prec -> Str -> (_,_ : PrecExp) -> PrecExp = \p,op,x,y ->
  {s = usePrec x p ++ op ++ usePrec y (nextPrec p) ;
   p = p
  } ;
```

Some expression linearizations

```
lin
  EMul _ _ = infixL 3 "*" ;
  EAdd _ _ = infixL 2 "+" ;
  ESub _ _ = infixL 2 "-" ;
```

Expressions: concrete syntax of JVM

```
lincat
  Exp = {s : Str} ;
  Typ = {s : Str ; t : TypIdent} ;
lin
  EAdd t _ = binopt "add" t.t ;    -- x ; y ; iadd ;
  ESub t _ = binopt "sub" t.t ;
  EMul t _ = binopt "mul" t.t ;
```

Auxiliary operations for JVM instructions

```
oper
  binopt : Str -> TypIdent -> SS -> SS -> SS = \op, t -> {s =
    x.s ++
    y.s ++
    typInstr op t
    } ;
  typInstr : Str -> TypIdent -> Str = \instr,t -> case t of {
    TIInt    => "i" + instr ;
    TIFloat  => "f" + instr ;
    TIDouble => "d" + instr
    } ;
```

Declarations: abstract syntax

Use higher-order abstract syntax

```
cat
  Stm ;
  Var Typ ;

fun
  Decl : (A : Typ) -> (Var A -> Stm) -> Stm ;
```

Using variables

```
fun
  Assign : (A : Typ) -> Var A -> Exp A -> Stm -> Stm ;

  EVar   : (A : Typ) -> Var A -> Exp A ;
```

Example of binding

```
int x ;        Decl TInt (\x ->
x = 5 ;          Assign TInt x (EInt 5) (
return x ;         Return TInt (EVar TInt x)))
```

Linearization of variable bindings

We assume η-long form of syntax trees.

We add a record field for each variable symbol

$$(\lambda x_0 \ \rightarrow \ \cdots \ \rightarrow \ \lambda x_n \ \rightarrow \ b)^{\circ} \ = \ b^{\circ} \ast\ast \{\$_0 = x_0^{\circ}; \ldots; \$_n = x_n^{\circ}\}$$

Variable bindings: an example record

The term

```
(\x -> Assign TInt x (EInt 5) (
          Return TInt (EVar TInt x)))
```

linearizes in C to

```
{s = ["x = 5 ; return x ;"] ; $0 = "x"}
```

Variable bindings: an example rule

The declaration function

```
fun Decl : (A : Typ) -> (Var A -> Stm) -> Stm ;
```

has in C the linearization rule

```
lin Decl typ stm = {s = typ.s ++ stm.$0 ++ ";" ++ stm.s} ;
```

A problem with higher-order abstract syntax

With normal scoping rules of type theory, we permit

```
int x ;        Decl int   (\x ->
float x ;      Decl float (\x ->
x = 3.14 ;     Assign float x (EFloat 3 14) ...))
```

which is not allowed in C.

Top-level program structure

Program = sequence of function definitions
(potentially recursive ones)

```
int factr (int n) { ... }
...
```

Bindings on two levels:

- function symbols: `factr`
- local parameters: `n`

Abstract syntax of program structure

```
cat
  Program ;
  ListTyp ;
  Body ListTyp ;
  Fun ListTyp Typ ;
fun
  Empty : Program ;
  Funct : (AS : ListTyp) -> (V : Typ) ->
              (Fun AS V -> Body AS) -> Program ;
  FunctNil : (V : Typ) ->
              Stm -> (Fun NilTyp V -> Program) -> Program ;
  RecOne  : (A : Typ) ->
              (Var A -> Stm) -> Program -> Body (ConsTyp A NilTyp) ;
  RecCons : (A : Typ) -> (AS : ListTyp) ->
              (Var A -> Body AS) -> Program -> Body (ConsTyp A AS) ;
```

Loops: JVM code generatoin

```
lincat
  Stm = {s,s2,s3 : Str} ; -- code, storage size, labels
lin
  While exp loop =
    let
      test = "TEST_" ++ loop.s2 ;
      end  = "END_"  ++ loop.s2
    in instrl (
        "label" ++ test ++ ";" ++
        exp.s ++
        "ifeq" ++ end ++ ";" ++
        loop.s ++
        "goto"  ++ test ++ ";" ++
        "label" ++ end
        ) ;
```

JVM code generation: the main problem

Variables are represented by stack addresses:

```
iload 1
istore 1
```

Solution: Symbolic JVM

Assembler format with variable symbols instead of stack addresses

- easy linearization to Symbolic JVM
- easy assembly of Symbolic JVM to Jasmin JVM

We wrote a Haskell program for the latter.

From symbolic JVM to Jasmin assembler

```
int x ;   alloc i x      ; x gets address 0
int y ;   alloc i y      ; y gets address 1
x = 5 ;   ldc 5          ldc 5
          istore x       istore 0
y = x ;   iload x        iload 0
          istore y       istore 1
```

Augustsson's Challenge 1

1. Write a printf function handling %d and %s.

```
fun Printf : (A : Typ) -> Exp A -> Stm -> Stm ;

lin Printf t e =
  continues ("printf" ++ paren (t.s2 ++ "," ++ e.s)) ;

lin Printf t e =
  instrc (e.s ++
          "runtime" ++ typInstr "printf" t.t ++ paren (t.s) ++ "V"
         ) ;
```

Code statistics

Code specific for this compiler

```
  51     368    1677 Imper.gf
  56     355    1764 ImperC.gf
  93     535    2870 ImperJVM.gf
 200    1258    6311 total
```

Code reusable in other compilers

```
  85     584    2624 ResImper.gf
  57     295    1835 Assemble.hs
```

What else we have achieved

Syntax editing (C and JVM in parallel!)

Decompilation (from Symbolic JVM)

Well-typedness of generated bytecode

Compiled parser (GF $\rightarrow$ BNFC $\rightarrow$ Happy/YACC)

What we cannot do

Non-compositional instruction selection

Prospects

Can this toy be extended to a tool?

I.e. a tool for implementing programming languages.