# Compiler Construction

## Exam

Wednesday, April 20, 2011, 14.00–17.00
BBL-061

---

- Please make sure that you write your name and student number on each sheet of paper that you hand in. On the first sheet of paper, indicate the total number of sheets handed in.
- This is a closed-book exam: it is not allowed to consult lecture notes, books, etc.
- Always explain the reasoning behind your answers, even when not explicitly asked for.
- Plan the making of the exam. Some questions involve coding; it is easy get stuck in a detail and spend too much time on it.
- For each fragment of attribute-grammar code, you may assume that the UUAG compiler is invoked with the `-d` flag, that is, corresponding Haskell data types are available.

---

# 1 SKI translation of Lambda-terms using AG

Appendix A provides an Attribute Grammar (AG) based implementation of a small compiler from lambda-calculus to SKI combinators. The source code is based on various parts of the course (AG programming, interpretation, SKI combinators), of which the main ingredients are pointed out here, and the missing parts are to be found in the appendix.

- **Source language: expressions / lambda-calculus**. The basic expression language:

```
data Exp
  | Var  n :: {String}
  | Int  i :: {Int}
  | Lam  n :: {String}
         e :: Exp
  | App  f :: Exp
         a :: Exp
```

- **Target langauge: SKI combinators**.

```
data SKI
  | S | K | I          -- basic combinators constants
  | A f :: SKI         -- combination via application
      a :: SKI
  | V n :: {String}    -- local/global reference
  | N i :: {Int}       -- builtin value
```

- **Translation** from source to target language. E.g.

$$\begin{array}{l}(\mathit{Exp\_Lam}\ \texttt{"x"}\ (\mathit{add}\ `\mathit{Exp\_App}`\ x\ `\mathit{Exp\_App}`\ x))\\ `\mathit{Exp\_App}`\\ (\mathit{add}\ `\mathit{Exp\_App}`\ \mathit{three}\ `\mathit{Exp\_App}`\ \mathit{three})\end{array}$$

  abstractly representing $(\lambda x \to x + x)\ (3 + 3)$ is translated to

$$(S\ (S\ (K\ \texttt{"add"})\ I)\ I\ (\texttt{"add"}\ 3\ 3))$$

  which itself yields 12 when further evaluated. (The missing definitions for e.g. *add* are found in Appendix A).

- **Execution** of the target language. The SKI representation is interpreted to yield a simplified (or normalized) SKI term:

$$\begin{array}{lll}
\mathit{s2v} :: \mathit{SKI} \to \mathit{SKI} & & \\
\quad \text{-- execution machinery} & & \\
\mathit{s2v}\ (\mathit{SKI_S}\ `\mathit{SKI\_A}`\ f\ `\mathit{SKI\_A}`\ g\ `\mathit{SKI\_A}`\ x) & = & \mathit{s2v}\ (\mathit{s2v}\ (f\ .@.\ x)\ .@.\ \mathit{s2v}\ (g\ .@.\ x)) \\
\mathit{s2v}\ (\mathit{SKI_K}\ `\mathit{SKI\_A}`\ y\ `\mathit{SKI\_A}`\ x\quad) & = & \mathit{s2v}\ y \\
\mathit{s2v}\ (\mathit{SKI_I}\ `\mathit{SKI\_A}`\ x\quad) & = & \mathit{s2v}\ x \\
\quad \text{-- primitives} & & \\
\mathit{s2v}\ (\mathit{SKI_V}\ \texttt{"add"}\ `\mathit{SKI\_A}`\ x\ `\mathit{SKI\_A}`\ y) & = & \mathit{SKI_N}\ (x' + y') \\
 & & \textbf{where}\ (\mathit{SKI_N}\ x') = \mathit{s2v}\ x \\
 & & \qquad\quad (\mathit{SKI_N}\ y') = \mathit{s2v}\ y \\
\quad \text{-- fallback/default} & & \\
\mathit{s2v}\ s & = & s
\end{array}$$

**Question 1.** The implementation (as explained in *A New Implementation Technique for Applicative Languages* by Turner) describes optimizations in terms of additional combinators $B$ and $C$, which are special cases of the $S$ combinator where it is known that the first respectively the second argument of $S$ does not need the last argument. Implement this. More concretely:

- Give the definition of the $B$ and $C$ combinators, and the optimization of $S$ in terms of $B$ and $C$ as described by Turners paper.
- Extend the definition of *SKI* and its execution in function *s2v*.
- Adapt the generation of *SKI* from `Exp` to optimize for the special cases described by Turners paper.

■

# 2 Code Generation

Consider the following program written in the imperative language that was discussed during the lecture on code generation,

```
var x;
var multiplier;
function fac(y)
{   if (y ≡ 0)
        return 1;
    else
    {   return y * multiplier * fac(y − 1);
    }
}
function main()
{   multiplier = 2;
    x = fac(3);
    print x;
},
```

and the associated scheme for generating assembly code for the Simple Stack Machine (SSM) discussed during the same lecture (and elaborated upon in Mini Project C, see Appendix B).

**Question 2.** Draw the stack layout constructed by the SSM at the point in the execution of the above program where the body of *fac* is entered for the second time and the execution of the **if** statement starts. Clearly indicate the locations of return addresses, saved mark-pointer addresses, static links, and identifiers. Also indicate the addresses pointed at by the values in the registers SP and MP. ■

**Question 3.** Give the generated SSM-code for the following fragments of the above program:

- The function *fac*.
- The fragment *fac*(3) of the body of *main*.

■

Suppose the language allows functions as parameters as well as return values, as in the following example locally declaring a function to be returned for use elsewhere:

```
function plusX(x)
{   function plusY(y)
    {   return x + y;
    }
    return plusY;
}
function main()
{   print plusX(3)(5);
}
```

**Question 4.** What is the runtime information by which a function must be represented when the program is executed? How does this interact with the use of a stack to store local variables and parameters? ■

# 3 Type Inference

Consider a small, implicitly typed functional programming language,

$$\begin{array}{rcll} x & \in & \mathbf{Var} & \text{variables} \\ t & \in & \mathbf{Tm} & \text{terms,} \end{array}$$

defined by

$$\begin{array}{rcl} t & ::= & n \mid \texttt{false} \mid \texttt{true} \mid x \mid \lambda x.\, t_1 \mid t_1\ t_2 \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni}, \\ & \mid & \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\ \mathbf{fi} \end{array}$$

and assume we subject it to the Hindley-Milner (HM) typing discipline. Its type language is then built from type variables,

$$\alpha \quad \in \quad \mathbf{TyVar} = \{\alpha, \beta, \gamma, \cdots\} \qquad \text{type variables,}$$

and stratified into types and type schemes, or monomorphic and polymorphic types respectively.

$$\begin{array}{rcll} \tau & \in & \mathbf{Ty} & \text{types} \\ \sigma & \in & \mathbf{TyScheme} & \text{type schemes,} \end{array}$$

given by

$$\begin{array}{rcl} \tau & ::= & \mathsf{Nat} \mid \mathsf{Bool} \mid \alpha \mid \tau_1 \to \tau_2 \\ \sigma & ::= & \tau \mid \forall \alpha.\, \sigma_1. \end{array}$$

Consider the adaption of Robinson's unification algorithm,

$$\mathcal{U} : \mathbf{Ty} \times \mathbf{Ty} \rightharpoonup \mathbf{TySubst},$$

that is used by algorithm W.

**Question 5.** Give the results of the following calls to $\mathcal{U}$:

(i) $\mathcal{U}(\alpha \to \beta \to \mathsf{Bool}\ ,\ \beta \to \mathsf{Nat} \to \alpha)$,

(ii) $\mathcal{U}(\alpha \to \mathsf{Bool} \to \beta\ ,\ \beta \to \alpha \to \gamma)$,

(iii) $\mathcal{U}(\alpha \to \mathsf{Bool} \to \beta\ ,\ \gamma \to \alpha)$,

(iv) $\mathcal{U}((\mathsf{Nat} \to \alpha) \to \beta \to \mathsf{Bool}\ ,\ \beta \to \alpha \to \gamma)$. ∎

Assume an environment $\Gamma$ in the type rules for the HM system with

$$\begin{array}{l} [\, \mathit{id} \mapsto \forall a.\, a \to a, \mathit{ii} \mapsto \mathsf{Nat} \to \mathsf{Nat}, \\ \ \mathit{const} \mapsto \forall a.\, \forall b.\, a \to b \to a, \\ \ \mathit{flip} \mapsto \forall a.\, \forall b.\, \forall c.\, (a \to b \to c) \to b \to a \to c \\ ] \subset \Gamma \end{array}$$

**Question 6.** In the HM system, give the derivation tree for:

$$\mathbf{let}\ x = \mathit{flip}\ \mathit{const}\ (\mathit{id}\ 3)\ \mathbf{in}\ x\ (\mathit{id}\ \texttt{true})$$

∎

The HM typing discipline cannot type

$$\begin{array}{ll} \textbf{let } f = \lambda i \rightarrow \mathit{const}\ (i\ 3)\ (i\ \texttt{true}) & \textbf{in} \\ \textbf{let } x = f\ \mathit{id} & \textbf{in } x \end{array}$$

although one would expect it to also type correctly because both program fragment variants can be rewritten into eachother. In the HM system only monomorphic types $\tau$ may participate in type inference, not polymorphic types $\sigma$, which only are allowed to exist in environments $\Gamma$. This causes the problem because then it is not possible to infer a polymorphic type for $f$'s parameter $i$. An obvious attempt to solve the problem therefore is to simply allow polymorphic types $\sigma$ to participate in type inference.

Given the following definitions:

$$\begin{array}{lll} \textbf{let } \mathit{f1} & = \lambda i \rightarrow \mathit{const}\ (i\ 3)\ (i\ \texttt{true}) & \textbf{in} \\ \textbf{let } \mathit{f2} & = \lambda i \rightarrow \mathit{id}\quad (i\ 3) & \textbf{in} \\ \textbf{let } \mathit{x11} & = \mathit{f1}\ \mathit{id} & \textbf{in} \\ \textbf{let } \mathit{x12} & = \mathit{f1}\ \mathit{ii} & \textbf{in} \\ \textbf{let } \mathit{x21} & = \mathit{f2}\ \mathit{id} & \textbf{in} \\ \textbf{let } \mathit{x22} & = \mathit{f2}\ \mathit{ii} & \textbf{in} \cdots \end{array}$$

**Question 7.** Assign the most general types to *f1* and *f2*, in that the greatest number of subsequent uses in the definitions of *x11*, *x12*, *x21*, and *x22* are type correct. Observe that there are more valid choices for the type of *f2*'s parameter $i$ than for *f1*'s parameter $i$. ∎

Such most general types are called *principal* types. The HM system guarantees that such principal types can always be inferred. However, when polymorphic types $\sigma$ are allowed to participate in type inference, principality breaks, exemplified by the following fragment:

$$\begin{array}{lll} \textbf{let } \mathit{choose} = \lambda x\ y \rightarrow \textbf{if} \cdots \textbf{then } x \textbf{ else } y \textbf{ fi} & \textbf{in} \\ \textbf{let } f \quad = \mathit{choose}\ \mathit{id} & \textbf{in} \cdots \end{array}$$

**Question 8.** There are two most general types for $f$ which do not have a more general type of which both are an instance (the requirement for principality). What are these types? Hint: a choice for the type of $f$'s parameter $y$ (from *choose*) can be made, similar to the previous question. Why is not one of these types more general than the other? ∎

# 4 Heap allocation & Garbage collection

Garbage collection is the process of reclaiming memory previously allocated on a heap and making it available for reuse. Conceptually the collection process consists of two phases: garbage *detection* and garbage *reclamation*.

In the following let *mem* be the memory used as a heap, an array of memory locations with valid indices $\in \{0\,..\,sz_m - 1\}$. Similary the root set *roots* (indices $\in \{0\,..\,sz_r - 1\}$) is an array containing pointers to live memory locations.

**Question 9.** Describe Mark-Sweep collection in terms of *mem* and *roots*, using pseudo-imperative code where appropriate. Describe the additionally required datastructures, the invariants to be maintained, and assumptions to be guaranteed by the program using the memory. ∎

Conceptually, garbage collection is simple; the problems lie in its engineering. Here problems related to the root set are considered. For any realistic system the root set will not be a single array, but will consist of various independent regions of memory used by a program: a stack, registers. It is essential that the root set holds exactly the values which must be detected for collection: too few means that program values will not be collected but still used (dangling reference), too many means that values will be retained but not used (space leak), bitpatterns not representing pointers to *mem* will create havoc in other unpredictable ways.

Very soon any language implementation puts values on a heap, pointed to from the stack, either because copying a pointer takes less execution time or a value must be available longer that the last-in-first-out (LIFO) behavior of a stack allows. In general, runtime systems and garbage collectors therefore need to manipulate various types of values, which we assume here to be one of (1) pointers to values on the heap (which must be collected), (2) Int values (which may not be collected), or (3) administrative values like saved programcounters (which are pointers, but not pointing the heap, may not be collected).

**Question 10.** A garbage collector must know which locations of the stack can be treated as a root. Design a solution to this problem: what additional datastructures are required at runtime? How would a garbage collector use these? When would such datastructures be constructed, at compile time, at runtime, or a combination of both? Better solutions are of course those who require little runtime overhead! ∎

# A SKI source code

The source code for testing, imports, etc:

```
imports
{
import UU.Pretty
}
{
infixl 9`Exp_App`, `SKI_A`, .@.
runExp :: Exp → IO ()
runExp e
   = do let t = wrap_ExpRoot (sem_ExpRoot $ ExpRoot_Root e) (Inh_ExpRoot)
            s = ski_Syn_ExpRoot t
        print $ "SKI:" >#< s
        print $ "Val:" >#< s2v s
main :: IO ()
main
   = do runExp t1
        runExp t2
add   = Exp_Var "add"
x     = Exp_Var "x"
three = Exp_Int  3
t1 = x
t2 = (Exp_Lam "x" (add `Exp_App` x `Exp_App` x))
     `Exp_App`
     (add `Exp_App` three `Exp_App` three)
}
```

The SKI combinator language and its interpretation:

```
  -- wrapping interface
wrapper SKIRoot
  -- convenience functions
{
(.@.) = SKI_A
v = SKI_V
i = SKI_I
k = SKI_K
s = SKI_S
n = SKI_N
}
  --
data SKIRoot
  | Root  s     :: SKI
  -- SKI
data SKI
  | S | K | I                -- basic combinators
  | A     f    :: SKI        -- combination via application
          a    :: SKI
  | V     n    :: {String}   -- local/global reference
  | N     i    :: {Int}      -- builtin value
deriving SKI : Show
  -- pretty printing over SKI
attr SKI SKIRoot syn pp :: {PP_Doc}
sem SKI
  | S     loc . pp                        = pp "S"
  | K     loc . pp                        = pp "K"
  | I     loc . pp                        = pp "I"
  | V     loc . pp                        = pp (show @n)
  | N     loc . pp                        = pp (show @i)
  | A     loc . pp                        = pp_block "(" ")" " " @as
  -- extract fun + args
attr SKI
  syn as :: {[PP_Doc]}
sem SKI
  | A     loc . as                        =  @f. as ++ [ @a. pp]
  | * - A lhs. as                         = [ @pp]
{
ppSKI e
  = pp_Syn_SKIRoot t
  where t = wrap_SKIRoot (sem_SKIRoot $ SKIRoot_Root e) (Inh_SKIRoot)
instance PP SKI where
  pp = ppSKI
}
  -- evaluation of SKI
{
s2v :: SKI → SKI
  -- execution machinery
s2v (SKI_S `SKI_A` f `SKI_A` g `SKI_A` x) = s2v (s2v (f .@. x) .@. s2v (g .@. x))
s2v (SKI_K `SKI_A` y `SKI_A` x          ) = s2v y
s2v (SKI_I `SKI_A` x                    ) = s2v x
  -- primitives
s2v (SKI_V "add" `SKI_A` x `SKI_A` y    ) = SKI_N (x' + y')
                                            where (SKI_N x') = s2v x
                                                  (SKI_N y') = s2v y
  -- fallback/default
s2v s                                     = s
}
```

The expression language Exp:

```
  -- wrapping interface
wrapper ExpRoot
  --
data ExpRoot
  | Root        e     :: Exp
  -- expression
data Exp
  | Var         n     :: {String}
  | Int         i     :: {Int}
  | Lam         n     :: {String}
                e     :: Exp
  | App         f     :: Exp
                a     :: Exp
  -- semantics: SKI
attr Exp ExpRoot syn ski :: SKI
sem Exp
  | Var         loc.ski  = v @n
  | Int         loc.ski  = n @i
  | App         loc.ski  = @f.ski .@. @a.ski
  | Lam         loc.ski  = @e.abs @n
  -- abstraction over a variable
attr Exp syn abs :: {String → SKI}
sem Exp
  | Var         lhs.abs = λn → if n ≡ @n then i else k .@. v @n
  | App         lhs.abs = λn → s .@. @f.abs n .@. @a.abs n
  | * - Var App lhs.abs = \_ → k .@. @ski
```

# B SSM abstract syntax

The *Instruction* datatype as used in miniproject C:

```
data Instruction
        -- Copying
    = Ldc   Const                -- Load constant.
    | LdcL  Label                -- Load label.
    | Lds   Offset               -- Load from stack.
    | Ldl   Offset               -- Load local.
    | Lda   Offset               -- Load via address.
    | Ldr   Register             -- Load register.
    | Ldrr  Register Register    -- Load register from register.
    | Sts   Offset               -- Store into stack.
    | Stl   Offset               -- Store local.
    | Sta   Offset               -- Store via address.
    | Str   Register             -- Store register.
        -- Convenience
    | Ajs   Offset               -- Adjust stack.
        -- Arithmetic
    | Add                        -- Addition.
    | Sub                        -- Subtraction.
    | Mul                        -- Multiplication.
    | Div                        -- Division.
    | Eq                         -- Test for equal.
    | Lt                         -- Test for less than.
    | Gt                         -- Test for greater than.
        -- Control
    | Bra   Label                -- Unconditional branch.
    | Brf   Label                -- Branch on false.
    | Jsr                        -- Jump to subroutine.
    | Ret                        -- Return.
        -- Other
    | Nop                        -- No operation.
    | Trap  Const                -- Trap to environment.
    | Halt                       -- Halt execution.
        -- Pseudoinstructions
    | Label Label                -- Label.
```