Automata and Formal Languages (10)

Email:	christian.urban at kcl.ac.uk
Office:	S1.27 (1st floor Strand Building)

Slides:	KEATS (also home work is there)

There are more problems, than there are programs.

There are more problems, than there are programs.

There must be a problem for which there is no program.

Last Week

$$\text{if } \varnothing \text{ does not occur in } r \quad \text{then} \quad L(r) \neq \{\}$$

holds, or equivalently

$$L(r) = \{\} \quad \text{implies} \quad \varnothing \text{ occurs in } r.$$

Last Week

if $\varnothing$ does not occur in r then $L(r) \neq \{\}$

holds, or equivalently

$L(r) = \{\}$ implies $\varnothing$ occurs in r.

$$
\begin{array}{lcl}
occurs(\varnothing) & \stackrel{\text{def}}{=} & true \\
occurs(\epsilon) & \stackrel{\text{def}}{=} & false \\
occurs(c) & \stackrel{\text{def}}{=} & false \\
occurs(r_1 + r_2) & \stackrel{\text{def}}{=} & occurs(r_1) \vee occurs(r_2) \\
occurs(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & occurs(r_1) \vee occurs(r_2) \\
occurs(r^*) & \stackrel{\text{def}}{=} & occurs(r)
\end{array}
$$

Functional Programming

```
def fib(n) = if n == 0 then 0
             else if n == 1 then 1
             else fib(n - 1) + fib(n - 2);

def fact(n) = if n == 0 then 1 else n * fact(n - 1);

def ack(m, n) = if m == 0 then n + 1
                else if n == 0 then ack(m - 1, 1)
                else ack(m - 1, ack(m, n - 1));

def gcd(a, b) = if b == 0 then a else gcd(b, a % b);
```

$$
\begin{array}{lcl}
\mathit{Exp} & \rightarrow & \mathit{Var} \mid \mathit{Num} \\
& \mid & \mathit{Exp} + \mathit{Exp} \mid \ldots \mid (\, \mathit{Exp}\,) \\
& \mid & \texttt{if}\ \mathit{BExp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \\
& \mid & \texttt{write}\ \mathit{Exp} \\
& \mid & \mathit{Exp}\ ;\ \mathit{Exp} \\
& \mid & \mathit{FunName}\ (\mathit{Exp},\ldots,\mathit{Exp}) \\
\mathit{BExp} & \rightarrow & \ldots \\
\mathit{Decl} & \rightarrow & \mathit{Def}\ ;\ \mathit{Decl} \\
& \mid & \mathit{Exp} \\
\mathit{Def} & \rightarrow & \texttt{def}\ \mathit{FunName}(x_1,\ldots,x_n) = \mathit{Exp}
\end{array}
$$

Abstract Syntax Tree

```
abstract class Exp
abstract class BExp
abstract class Decl

case class
  Def(name: String, args: List[String], body: Exp)
                                          extends Decl
case class Main(e: Exp) extends Decl

case class Call(name: String, args: List[Exp]) extends Exp
case class If(a: BExp, e1: Exp, e2: Exp) extends Exp
case class Write(e: Exp) extends Exp
case class Var(s: String) extends Exp
case class Num(i: Int) extends Exp
case class Aop(o: String, a1: Exp, a2: Exp) extends Exp
case class Sequ(e1: Exp, e2: Exp) extends Exp

case class Bop(o: String, a1: Exp, a2: Exp) extends BExp
```

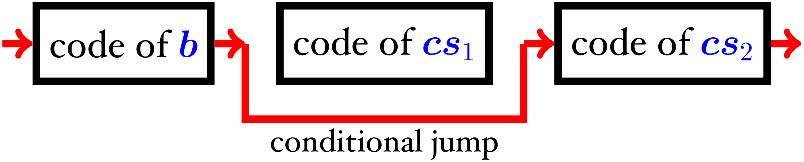
Mathematical Functions

Compilation of some mathematical functions:

$$
\begin{array}{lcl}
\texttt{Aop("+", a1, a2)} & \Rightarrow & \texttt{...iadd} \\
\texttt{Aop("-", a1, a2)} & \Rightarrow & \texttt{...isub} \\
\texttt{Aop("*", a1, a2)} & \Rightarrow & \texttt{...imul} \\
\texttt{Aop("/", a1, a2)} & \Rightarrow & \texttt{...idiv} \\
\texttt{Aop("\%", a1, a2)} & \Rightarrow & \texttt{...irem}
\end{array}
$$

Boolean Expressions

Compilation of boolean expressions:




```
Bop(”==”, a1, a2)  ⇒  ...if_icmpne...
Bop(”!=”, a1, a2)  ⇒  ...if_icmpeq...
Bop(”<”, a1, a2)   ⇒  ...if_icmpge...
Bop(”<=”, a1, a2)  ⇒  ...if_icmpgt...
```

Sequences

Compiling `arg1 ; arg2`:

```
...arg1...
pop
...arg1...
```

Write

Compiling `write(arg)`:

```
...arg...
dup
invokestatic XXX/XXX/write(I)V
```

```
case Write(a1) => {
    compile_exp(a1, env) ++
    List("dup\n",
         "invokestatic XXX/XXX/write(I)V\n")
  }
```

Functions

```
.method public static write(I)V
   .limit locals 5
   .limit stack 5
   iload 0
   getstatic java/lang/System/out Ljava/io/PrintStream;
   swap
   invokevirtual java/io/PrintStream/println(I)V
   return
.end method
```

We will need for definitions

```
.method public static f (I...I)I
  .limit locals ??
  .limit stack ??
  ??
.end method
```

Stack Estimation

```
def max_stack_exp(e: Exp): Int = e match {
  case Call(_, args) => args.map(max_stack_exp).sum
  case If(a, e1, e2) => max_stack_bexp(a) +
   (List(max_stack_exp(e1), max_stack_exp(e1)).max)
  case Write(e) => max_stack_exp(e) + 1
  case Var(_) => 1
  case Num(_) => 1
  case Aop(_, a1, a2) =>
   max_stack_exp(a1) + max_stack_exp(a2)
  case Sequ(e1, e2) =>
   List(max_stack_exp(e1), max_stack_exp(e2)).max
}

def max_stack_bexp(e: BExp): Int = e match {
  case Bop(_, a1, a2) =>
   max_stack_exp(a1) + max_stack_exp(a2)
}
```

Successor

```
.method public static suc(I)I
.limit locals 1
.limit stack
  iload 0
  ldc 1
  iadd
  ireturn
.end method
```

```
def suc(x) = x + 1;
```

Addition

```
.method public static add(II)I
.limit locals 2
.limit stack 4
  iload 0
  ldc 0
  if_icmpne If_else_2
  iload 1
  goto If_end_3
If_else_2:
  iload 0
  ldc 1
  isub
  iload 1
  invokestatic defs/defs/add(II)I
  invokestatic defs/defs/suc(I)I
If_end_3:
  ireturn
.end method
```

```
def add(x, y) =
    if x == 0 then y
    else suc(add(x - 1, y));
```

Factorial

```
.method public static facT(II)I
.limit locals 2
.limit stack 4
  iload 0
  ldc 0
  if_icmpne If_else_2
  iload 1
  goto If_end_3
If_else_2:
  iload 0
  ldc 1
  isub
  iload 0
  iload 1
  imul
  invokestatic fact/fact/facT(II)I
If_end_3:
  ireturn
.end method
```

```
def facT(n, acc) =
  if n == 0 then acc
  else facT(n - 1, n * acc);
```

```
.method public static facT(II)I
.limit locals 2
.limit stack 4
facT_Start:
  iload 0
  ldc 0
  if_icmpne If_else_2
  iload 1
  goto If_end_3
If_else_2:
  iload 0
  ldc 1
  isub
  iload 0
  iload 1
  imul
  istore 1
  istore 0
  goto facT_Start
If_end_3:
  ireturn
.end method
```

```
def facT(n, acc) =
  if n == 0 then acc
  else facT(n - 1, n * acc);
```

Tail Recursion

A call to `f(args)` is usually compiled as

```
args onto stack
invokestatic  .../f
```

Tail Recursion

A call to `f(args)` is usually compiled as

```
args onto stack
invokestatic  .../f
```

A call is in tail position provided:

- `if Bexp then Exp else Exp`
- `Exp ; Exp`
- `Exp op Exp`

then a call `f(args)` can be compiled as

```
prepare environment
jump to start of function
```

Tail Recursive Call

```
def compile_expT(a: Exp, env: Mem, name: String): Instrs =
  ...
  case Call(n, args) => if (name == n)
  {
    val stores = args.zipWithIndex.map
       { case (x, y) => "istore " + y.toString + "\n" }
    args.flatMap(a => compile_expT(a, env, "")) ++
    stores.reverse ++
    List ("goto " + n + "_Start\n")
  }
  else
  {
    val is = "I" * args.length
    args.flatMap(a => compile_expT(a, env, "")) ++
    List ("invokestatic XXX/XXX/" + n + "(" + is + ")I\n")
  }
```

There are more problems, than there are programs.

There must be a problem for which there is no program.

Subsets

$A \subseteq B$ and $B \subseteq A$

then $A = B$

Injective Function

f is an injective function iff

$$\forall xy.\ f(x) = f(y) \Rightarrow x = y$$

Cardinality

$$|A| \stackrel{\text{def}}{=} \text{“how many elements”}$$

$$A \subseteq B \Rightarrow |A| \leq |B|$$

Cardinality

$|A| \stackrel{\text{def}}{=}$ "how many elements"

$A \subseteq B \Rightarrow |A| \leq |B|$

if there is an injective function $f : A \rightarrow B$ then $|A| \leq |B|$

Natural Numbers

$$\mathbb{N} \stackrel{\text{def}}{=} \{0, 1, 2, 3, \ldots\ldots\}$$

Natural Numbers

$$\mathbb{N} \stackrel{\text{def}}{=} \{0, 1, 2, 3, \ldots\ldots\}$$

A is **countable** iff $|A| \leq |\mathbb{N}|$

First Question

$$|\mathbb{N} - \{0\}| \quad ? \quad |\mathbb{N}|$$

$\geq$ or $\leq$ or $=$

$$|\mathbb{N} - \{0, 1\}| \quad ? \quad |\mathbb{N}|$$

$$|\mathbb{N} - \{0, 1\}| \quad ? \quad |\mathbb{N}|$$

$$|\mathbb{N} - \mathbb{O}| \quad ? \quad |\mathbb{N}|$$

$\mathbb{O} \stackrel{\text{def}}{=}$ odd numbers $\{1, 3, 5......\}$

$$|\mathbb{N} - \{0, 1\}| \quad ? \quad |\mathbb{N}|$$

$$|\mathbb{N} - \mathbb{O}| \quad ? \quad |\mathbb{N}|$$

$\mathbb{O} \stackrel{\text{def}}{=}$ odd numbers $\{1, 3, 5......\}$
$\mathbb{E} \stackrel{\text{def}}{=}$ even numbers $\{0, 2, 4......\}$

$$|\mathbb{N} \cup -\mathbb{N}| \quad ? \quad |\mathbb{N}|$$

$\mathbb{N} \stackrel{\text{def}}{=}$ positive numbers $\{0, 1, 2, 3, \ldots\ldots\}$

$-\mathbb{N} \stackrel{\text{def}}{=}$ negative numbers $\{0, -1, -2, -3, \ldots\ldots\}$

A is countable if there exists an injective $f : A \to \mathbb{N}$

A is uncountable if there does not exist an injective $f : A \to \mathbb{N}$

countable: $|A| \leq |\mathbb{N}|$
uncountable: $|A| > |\mathbb{N}|$

A is countable if there exists an injective $f : A \to \mathbb{N}$

A is uncountable if there does not exist an injective $f : A \to \mathbb{N}$

countable: $|A| \leq |\mathbb{N}|$
uncountable: $|A| > |\mathbb{N}|$

Does there exist such an A ?

Halting Problem

Assume a program $\boldsymbol{H}$ that decides for all programs $\boldsymbol{A}$ and all input data $\boldsymbol{D}$ whether

- $\boldsymbol{H}(\boldsymbol{A}, \boldsymbol{D}) \stackrel{\text{def}}{=} 1$ iff $\boldsymbol{A}(\boldsymbol{D})$ terminates
- $\boldsymbol{H}(\boldsymbol{A}, \boldsymbol{D}) \stackrel{\text{def}}{=} 0$ otherwise

Halting Problem (2)

Given such a program H define the following program C: for all programs A

- $C(A) \overset{\text{def}}{=} 0$ iff $H(A, A) = 0$
- $C(A) \overset{\text{def}}{=}$ loops otherwise

Contradiction

$H(C, C)$ is either 0 or 1.

- $H(C, C) = 1 \stackrel{\text{def}\,H}{\Rightarrow} C(C)\downarrow \stackrel{\text{def}\,C}{\Rightarrow} H(C, C) = 0$
- $H(C, C) = 0 \stackrel{\text{def}\,H}{\Rightarrow} C(C)$ loops $\stackrel{\text{def}\,C}{\Rightarrow}$ $H(C, C) = 1$

Contradiction in both cases. So H cannot exist.