koka

A language with effect inference

Daan Leijen
Microsoft Research

http://www.rise4fun.com/koka/tutorial

daan@microsoft.com

Koka main features

- Strict evaluation (as in ML and F#)
- Separation of *pure* vs *side-effecting* computation through an effect system (as in Haskell)
- Familiar JavaScript-like syntax
- Function oriented

Effects and reasoning

- Is `e*0 == 0 ?` And `x*0 ?`
- Can `f(x)`, `g(x)` be done in parallel ?
- Is `f(e,e)` the same as `let x = e in f(x,x) ?`
- ….

Comes up all the time in practice: Parallelism, Tier-splitting, LINQ queries, HADOOP tasks, Sandboxing, STM, Stateless cloud services, etc.

Make effects explicit

- Just like **types** are useful to programmers to clarify and structure their code,
  effects should be part of the type signature to help reasoning and understanding:

```
(*)   : (int,int)      -> total int
(/)   : (int,int)      -> exn int
exec  : (turingMach)   -> div bool
(!)   : (ref<h,a>)     -> read<h> a
print : (string)       -> io ()
```

Effects are inferred

- Here is a nice total function:

```
function sqr1( x : int ) : total int
{
  return x*x
}
```

- If we add a print statement, the type is:

```
function sqr2( x : int ) : io int
{
  print( "not so secret side-effect" )
  return x*x
}
```

Effects and denotational semantics

- The effect annotations are not just extra ‘tags’ on the type: in essence, the type with an effect tells us exactly what the signature is of the (mathematical) semantic function that models this program:

$$
\begin{aligned}
[\![\texttt{int -> total int}]\!] &= \mathbb{Z} \rightarrow \mathbb{Z} \\
[\![\texttt{int -> exn int}]\!] &= \mathbb{Z} \rightarrow (\mathbb{Z}+1) \\
[\![\texttt{int -> read<}h\texttt{> int}]\!] &= heap \times \mathbb{Z} \rightarrow \mathbb{Z} \\
[\![\texttt{int -> <st<}h\texttt{>,div> int}]\!] &= heap \times \mathbb{Z} \rightarrow (heap \times \mathbb{Z})_{\bot}
\end{aligned}
$$

Challenges of effect inference:

- Combining type inference with polymorphic effects is a challenge:
  1. Inference generally becomes incomplete or undecidable with subtyping or type operators
  2. Effect types can easily become cumbersome to read or annotate for programmers.

Polymorphic effects

- Map a function over a list:

```
function map( f : (a) -> e b, xs : list<a> ) : e list<b>
{
  match(xs) {
    nil -> nil
    cons(x,xx) -> cons(f(x),map(f,xx))
  }
}
```

- Since map uses inductive recursion, it has no effect itself.

More challenging example

- Consider the while function which takes a predicate and action as arguments:

```
function while( pred, action )
{
  if (pred()) {
    action();
    while(pred,action);
  }
}
```

Using type operators?

- We may consider a + operation that combines different effects:

```
(() -> e1 bool, () -> e2 ()) -> (e1 + e2 + div) ()
```

- Unfortunately, that poses severe problems for inference. Suppose we need to unify:

```
e1 + e2  ~  div + exn + read<h>
```

- We cannot solve such constraints and making them explicit leads to unreadable types

Using sub-types?

- The type for while would become:

```
(e1 <= e3, e2 <= e3, div <= e3) =>
     ( () -> e1 bool, () -> e2 () ) -> e3 ()
```

- Again, the type becomes quite difficult.
- Requires full semi-lattice of effects
- This approach has been described in a tech report – and some simplications can be made:

```
(div <= e) =>  (() -> e bool, () -> e () ) -> e ()
```

Our approach: row-polymorphism

- We use a row (or set) of effect labels.
  - The empty effect is `<>`
  - An effect can be extended as `<div|e>`
  - Sugar: `<div,exn> == <div|<exn|<>>>`
  - Duplicates are ok: `<div|<div|e>> != <div|e>`

- The type for `while` becomes:

```
(() -> <div|e> bool, () -> <div|e> ()) -> <div|e> ()
```

- Simple types using regular unification.

Is the type for while too restrictive?

- Consider

```
while { odd(random-int()) }
{
  error("raise an exception")
}
```

- For the predicate we infer <ndet|*e1*>
- For the body we infer <exn|*e2*>
- Both unify to <exn|ndet|*e3*> and now it fits the signature of while

```
(() -> <div|e> bool, () -> <div|e> ()) -> <div|e> ()
```

Declarative type rules

$$(\text{VAR})\quad \frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma \mid \epsilon}$$

$$(\text{LAM})\quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \mid \epsilon_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \rightarrow \epsilon_2\ \tau_2 \mid \epsilon}$$

$$(\text{APP})\quad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \epsilon\ \tau \mid \epsilon \qquad \Gamma \vdash e_2 : \tau_2 \mid \epsilon}{\Gamma \vdash e_1\, e_2 : \tau \mid \epsilon}$$

$$(\text{SEQ})\quad \frac{\Gamma \vdash e_1 : \tau_1 \mid \epsilon \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \mid \epsilon}{\Gamma \vdash (x \leftarrow e_1; e_2) : \tau_2 \mid \epsilon}$$

$$(\text{LET})\quad \frac{\Gamma \vdash e_1 : \sigma \mid \langle\rangle \qquad \Gamma, x : \sigma \vdash e_2 : \tau \mid \epsilon}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau \mid \epsilon}$$

$$(\text{GEN})\quad \frac{\Gamma \vdash e : \tau \mid \langle\rangle \qquad \overline{\nu} \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash e : \forall \overline{\nu}.\, \tau \mid \langle\rangle}$$

$$(\text{INST})\quad \frac{\Gamma \vdash e : \forall \overline{\nu}.\, \tau \mid \epsilon}{\Gamma \vdash e : [\overline{\nu} := \overline{\tau}]\tau \mid \epsilon}$$

Effects in Koka

- It is easy to define effects:

```
type exn       : !
type div       : !
type ndet      : !
type read<h>   : H -> !
type write<h>  : H -> !
type global    : H

alias pure      = <div,exn>
alias st<h>     = <read<h>,write<h>>
alias io        = <st<global>,div,exn,ndet>
```

- Easy to add effects like `client-io` / `server-io` (for tier-splitting). Or `blocking`, or …

Catching exceptions

- When we catch exceptions, the exn effect can be discarded:

```
forall<e,a> ( action: () -> <exn|e> a,
            , handler: exception -> e a)
            -> e a
```

Duplicate labels:
<exn|<exn|e2>>

e can unify with <exn|e2>

Isolated state

- Heap operations are parameterized by a heap parameter *h*.
- Just like in the Haskell ST monad, we can use the equivalent of `runST` to turn stateful operations into pure operations again using a rank-2 polymorphic type:

```
runST: (forall<h> () -> <st<h>|e> a) -> e a
```

- This is automatically done by Koka if applicable at generalizations

Isolated state

- Fibonacci is a total function:

```
function fib( n : int ) : total int
{
  var x := 0
  var y := 1
  repeat(n) {
    y0 = y
    y := x+y
    x := y0
  }
  return x
}
```

- But the body has effect `st<h>`

Similar to `runST` in Haskell

Recursion and state

- The following function never terminates but there is no (syntactic) recursion:

```
function non-terminating() : div int
{
  r = ref(id)  // assign the identity function
  fun foo() {
    (!r)()     // call
  }
  r := foo     // ass    foo itself to r
  foo()        // .. and
}
```

```
r :ref<h,() -> <read<h>|e> ()>
```

```
foo : () -> <read<h>|e> ()
```

- Solution: we add an extra constraint,

(!) :

```
(ref<h,a>) -> <read<h>|e> a with hdiv<h,a,e>
```

```
() -> <read<h>|e> ()>
```

Questions?

- Try it out on the web: http://www.rise4fun.com/koka/tutorial
- Still under development but its getting there.

Hello world

- Familiar syntax.
- Layout rule: everything aligned between braces ({ and }) is semicolon separated.

```
function main()
{
  print("Hello world!") // print output
}
```

Caesar encoding

```
function caesar( s : string ) : string
{
  function encode-char(c) {
    if (!c.is-lower) return c
    base = (c - 'a').to-int
    rot  = (base + 3) % 26
    return (rot.to-char + 'a')
  }

  s.map(encode-char)
}
```

Dot notation

- Koka is “function oriented”

```
s.map(f) == map(s,f)
```

- Nice for chaining:

```
s.caesar.map(to-upper).print
```

- Everything is just a function!
- And all functions are ‘extension methods’

Function arguments may follow the calling parenthesis

```
function caesar( s : string ) : string
{
  s.map() fun(c) {
    if (!c.is-lower) return c
    base = (c - 'a').to-int
    rot  = (base + 3) % 26
    return (rot.to-char + 'a')
  }
}
```

While is not built-in

- While is a function taking two actions as arguments:

```
function print10()
{
  var i := 0
  while( fun(){ i < 10 }, fun(){
    print(i)
    i := i+1
  })
}
```

Inductive types

```
type void { }

type unit { unit }

type bool { false; true }

type color {
  red
  green
  blue
}

type maybe<a> {
  nothing
  just( value : a )
}
```

- A struct is just a type with a single constructor of the same name. eg: `struct unit`

Optional and named parameters

- Named parameters can be used in any order:

```
function world() {
  "hi world".substring( len=5, start=3 )
}
```

- Optional parameters have a default value:

```
function sublist( xs : list<a>, start : int,
                  len : int = xs.length ) : list<a>
{
  if (start <= 0) return xs.take(len)
  match(xs) {
    nil -> nil
    cons(_,xx) -> xx.sublist(start-1, len)
  }
}
```

Structs

- Struct declarations are quite like function declarations:

```
struct person( age : int, name : string
             , realname : string = name )
```

- Koka generates a constructor function with the same name for each struct.

```
person( name = "Lady Gaga", age = 25
  , realname = "Stefani Joanne Angelina Germanotta")
```

- And an accessor function for each field:

```
person(25,"Lady Gaga").realname
```

Immutable by default

- Structs are immutable by default: changing a field usually copies a struct:

```
function birthday( p : person ) {
   p( age = p.age + 1 )
}
```

- When applying to a struct, Koka invokes a copy constructor, basically defined as:

```
function copy( p, age = p.age, name = p.name
             , rname = p.realname )
{
   person(age, name, rname)
}
```

- No special rules: just functions!