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 \( \text{let } x = e \text{ in } f(x,x) \)?
- ....

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:

```plaintext
(\*) : (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:

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

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

```c
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:

\[
\begin{align*}
\text{[int -> total int]} & \quad = \quad \mathbb{Z} \rightarrow \mathbb{Z} \\
\text{[int -> exn int]} & \quad = \quad \mathbb{Z} \rightarrow (\mathbb{Z}+1) \\
\text{[int -> read<h> int]} & \quad = \quad \text{heap} \times \mathbb{Z} \rightarrow \mathbb{Z} \\
\text{[int -> <st<h>,div> int]} & \quad = \quad \text{heap} \times \mathbb{Z} \rightarrow (\text{heap} \times \mathbb{Z})_\perp
\end{align*}
\]
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:

```plaintext
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:

```javascript
function while( pred, action )
{
    if (pred()) {
        action();
        while(pred,action);
    }
}
```
Using type operators?

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

\[
(() \to e_1 \text{ bool}, (\) \to e_2 (\)) \to (e_1 + e_2 + \text{div}) ()
\]

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

\[
e_1 + e_2 \sim \text{div} + \text{exn} + \text{read}<h>\]

• We cannot solve such constraints and making them explicit leads to unreadable types
Using sub-types?

• The type for while would become:

\[(e_1 \leq e_3, e_2 \leq e_3, \text{div} \leq e_3) \Rightarrow (() \rightarrow e_1 \text{ bool}, () \rightarrow e_2 () ) \rightarrow e_3 ()\]

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

\[(\text{div} \leq e) \Rightarrow (() \rightarrow e \text{ bool}, () \rightarrow e () ) \rightarrow e ()\]
Our approach: row-polymorphism

• We use a row (or set) of effect labels.
  ▪ The empty effect is $\langle\rangle$
  ▪ An effect can be extended as $\langle\text{div}|\text{e}\rangle$
  ▪ Sugar: $\langle\text{div,exn}\rangle =\langle\text{div}|\langle\text{exn}|\langle\rangle\rangle\rangle$
  ▪ Duplicates are ok: $\langle\text{div}|\langle\text{div}|\text{e}\rangle\rangle \neq \langle\text{div}|\text{e}\rangle$

• The type for while becomes:

  $((()) \rightarrow \langle\text{div}|\text{e}\rangle \text{ bool}, () \rightarrow \langle\text{div}|\text{e}\rangle (()) \rightarrow \langle\text{div}|\text{e}\rangle ()$ 

• 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

(VAR) \[
\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma | \epsilon} \]

(LAM) \[
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 | \epsilon_2}{\Gamma \vdash \lambda x. e : \tau_1 \to \epsilon_2 \tau_2 | \epsilon} \]

(APP) \[
\frac{\Gamma \vdash e_1 : \tau_2 \to \epsilon \tau | \epsilon \quad \Gamma \vdash e_2 : \tau_2 | \epsilon}{\Gamma \vdash e_1 e_2 : \tau | \epsilon} \]

(SEQ) \[
\frac{\Gamma \vdash e_1 : \tau_1 | \epsilon \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 | \epsilon}{\Gamma \vdash (x \leftarrow e_1 ; e_2) : \tau_2 | \epsilon} \]

(LET) \[
\frac{\Gamma \vdash e_1 : \sigma | \langle \rangle \quad \Gamma, x : \sigma \vdash e_2 : \tau | \epsilon}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau | \epsilon} \]

(GEN) \[
\frac{\Gamma \vdash e : \tau | \langle \rangle \quad \nu \not\in \text{ftv}(\Gamma)}{\Gamma \vdash e : \forall \nu. \tau | \langle \rangle} \]

(INST) \[
\frac{\Gamma \vdash e : \forall \nu. \tau | \epsilon}{\Gamma \vdash e : [\nu := \bar{\tau}] \tau | \epsilon} \]
Effects in Koka

• It is easy to define effects:

```plaintext
  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:

```plaintext
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:

\[
\text{runST: } (\forall h \ (\_ \to \textit{st}_h \mid e \ a)) \to e \ a
\]

• This is automatically done by Koka if applicable at generalizations
Isolated state

• Fibonacci is a total function:

```cpp
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:

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

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

• Solution: we add an extra constraint,

```plaintext
(!): (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.

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

\[
s\text{.map}(f) == \text{map}(s,f)
\]

• Nice for chaining:

\[
s\text{.caesar.map(to-upper).print}
\]

• Everything is just a function!
• And all functions are ‘extension methods’
Function arguments may follow the calling parenthesis

```rust
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:

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

A struct is just a type with a single constructor of the same name. e.g: `struct unit`
Optional and named parameters

• Named parameters can be used in any order:

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

• Optional parameters have a default value:

```javascript
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:

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

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

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

- No special rules: just functions!