Kiama: Domain-Specific Languages for Language Implementation in Scala

Anthony Sloane

Anthony.Sloane@mq.edu.au
inkytonik@gmail.com

@inkytonik

Matthew Roberts

Matthew.Roberts@mq.edu.au
altmattr@gmail.com

@altmattr

Programming Languages Research Group
Department of Computing
Macquarie University

May 15, 2013

Language processing problems

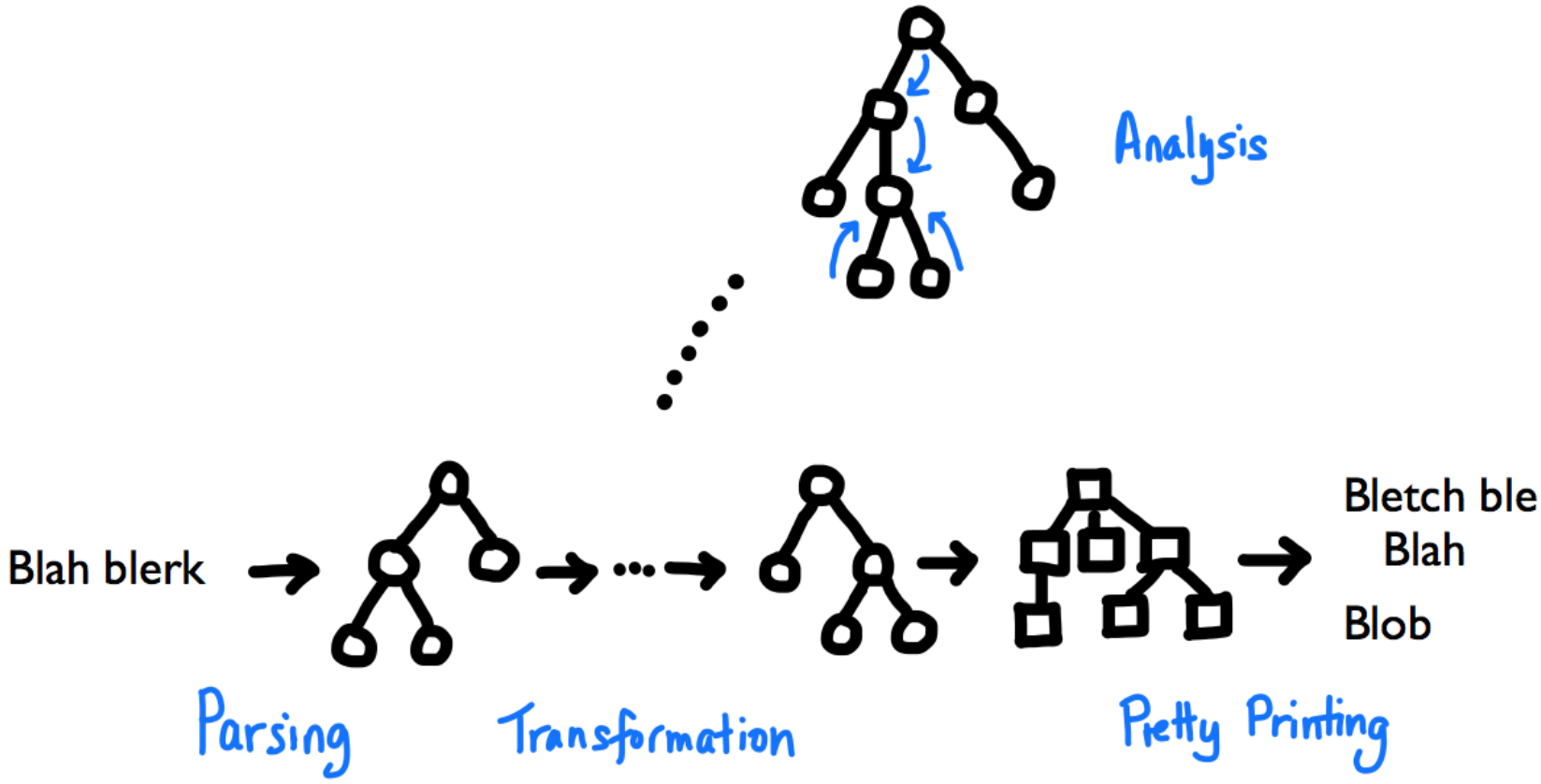

Figure : The big picture

Kiama

A collection of Scala-based *internal domain-specific languages* for solving problems in the domain of language processing.

Problem	*Domain-specific language*
Text to structure	Context-free grammars
Structure representation	Algebraic data types
Transformation	Term rewriting
Analysis	Attribute grammars
Structure to text	Pretty printing

Lambda calculus REPL

```
> 1 + 2
3 : Int

> (\x : Int . x + 1) 4
5 : Int

> let i : Int = ((\x : Int . x + 1) 4) in i + i
10 : Int

> \x : Int . x + y
1.16: y: unknown variable

> \x : Int . x 4
1.12: x: non-function applied

> (\f : Int -> Int . f 4) 5
1.25: 5: expected Int -> Int, got Int
```

Abstract Syntax Trees

```
let i : Int = ((\x : Int . x + 1) 4) in i + i
```

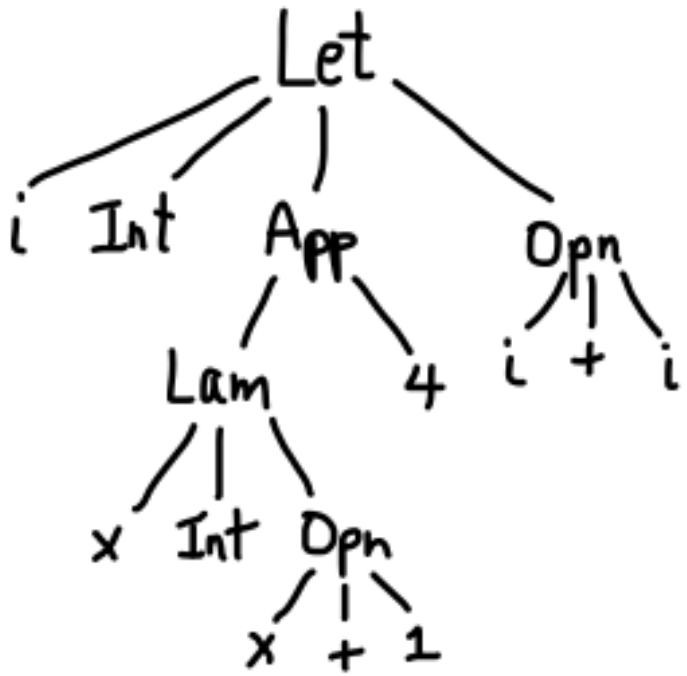

Figure : The data is naturally represented by a hierarchical data structure

Evaluation: beta reduction rule

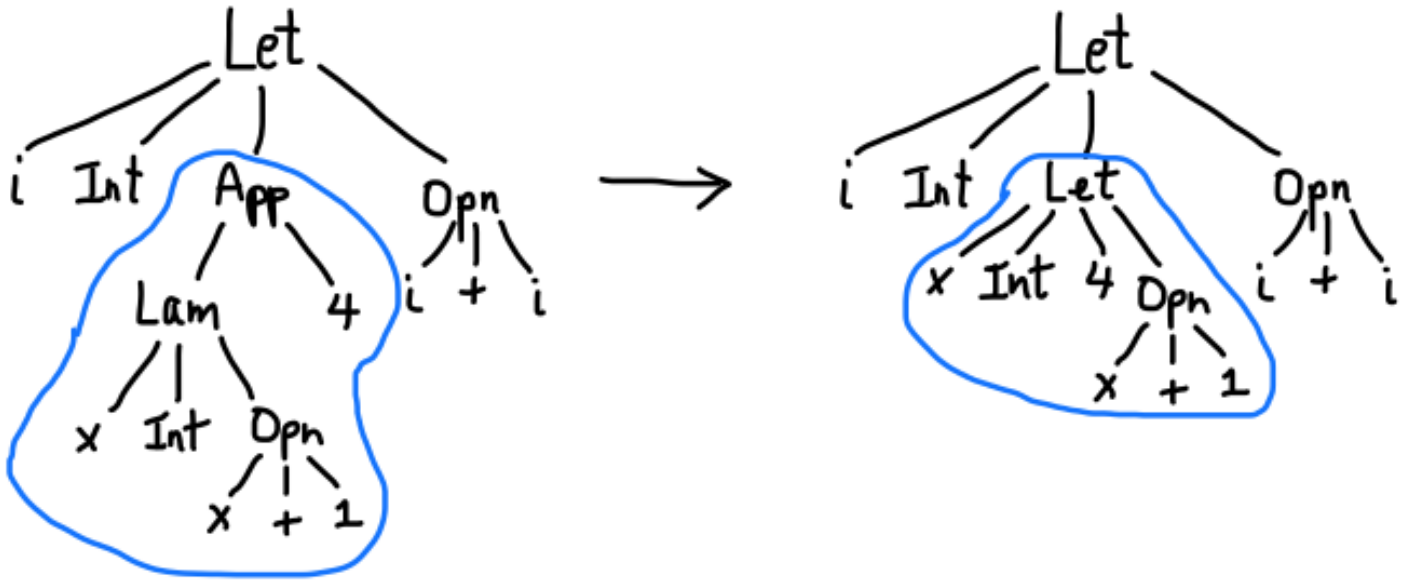

Figure : Replace an application by a substitution into the body

```
val beta =
  rule {
    case App (Lam (x, t, e1), e2) =>
      Let (x, t, e2, e1)
  }
```

Evaluation: substitution rule

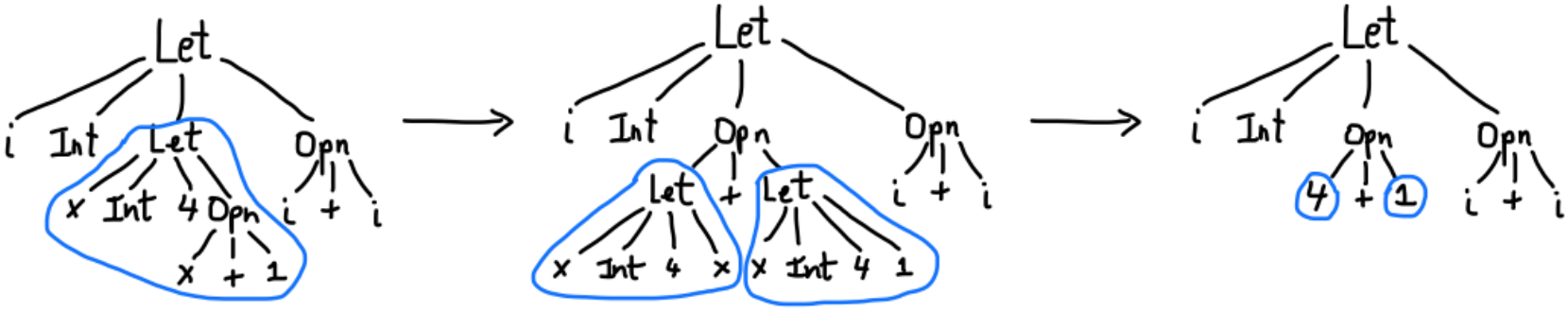

Figure : Push substitutions into operations, replace variables

```
val substitution =
    rule {
        case Let (x, t, e, Opn (l, op, r)) =>
          Opn (Let (x, t, e, l), op, Let (x, t, e, r))
        case Let (x, _, e, v @ Var (y))     =>
          if (x == y) e else v
        case Let (_, _, _, n : Num)         => n
        ...
     }
```

Choosing between rewrite strategies

```
val onestep : Strategy =
  beta <+ arithop <+ substitution
```

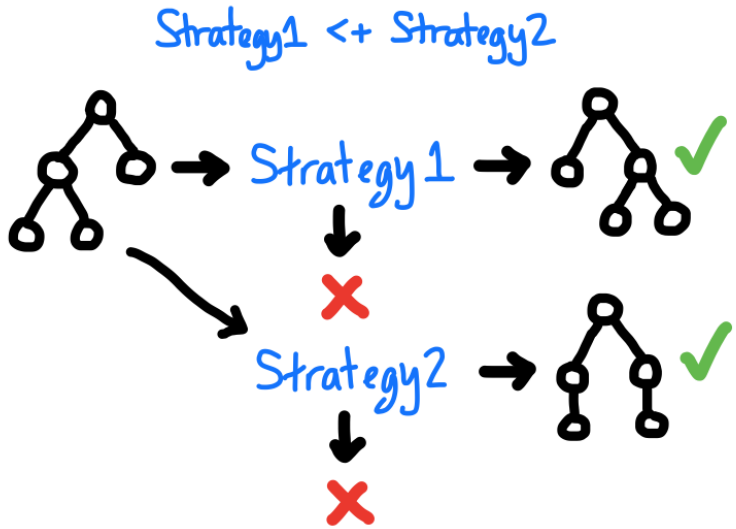

Figure : A choice succeeds iff one of its alternatives succeeds

Generic traversal: some

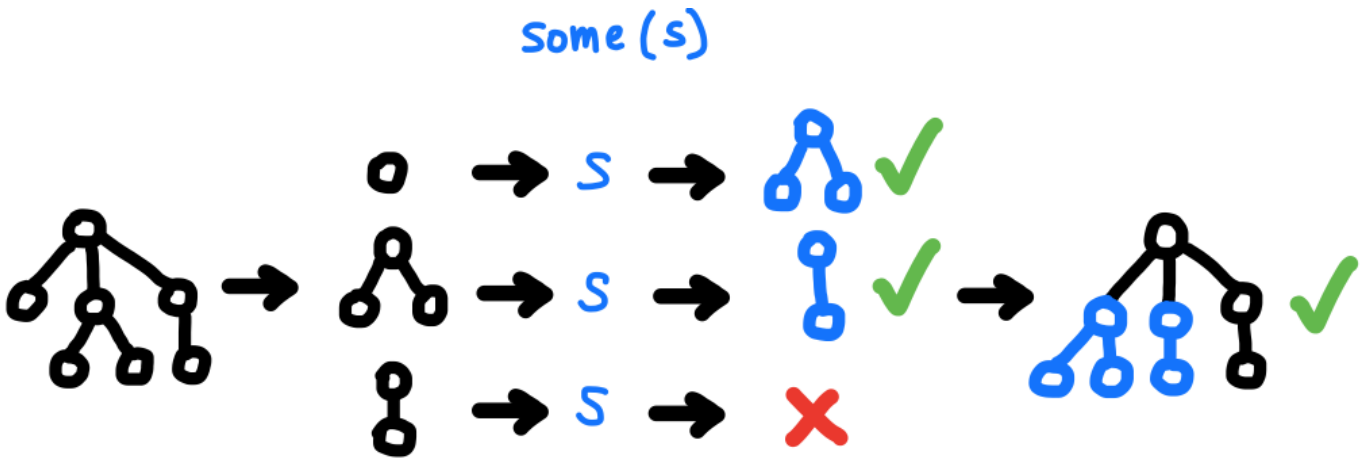

Figure : some (s) succeeds if s succeeds at least one of the children

Building up a traversal

- At the root

```
onestep
```

- At one or more children of the root

```
some (onestep)
```

- At one or more children of the root, or if that fails, at the root

```
some (onestep) <+ onestep
```

- At the topmost nodes at which it succeeds

```
lazy val inner = some (inner) <+ onestep
```

Type checking: attributes

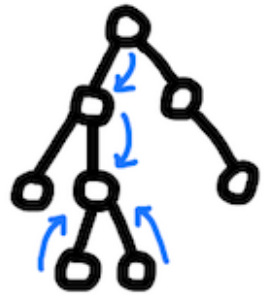

Figure : Attributes are memoised functions of nodes

```
val envOf : Exp => List[(String,Type)] =
  attr {
    ... cases go here ...
  }

val typeOf : Exp => Type =
  attr {
    ... cases go here ...
  }
```

Environments

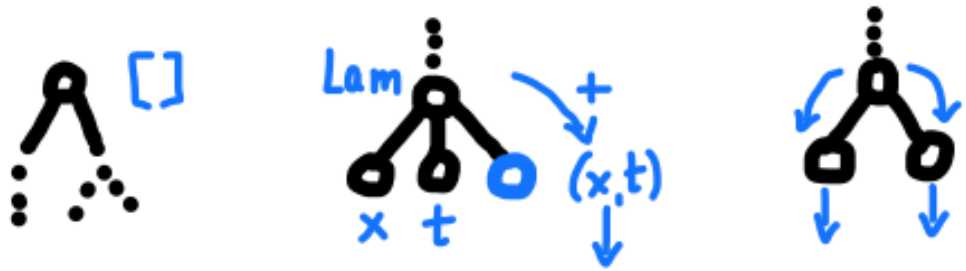

Figure : Propagate downward, gather bindings at lambdas and lets

```
case n if n.isRoot                  => List ()
case n =>
  n.parent match {
    case p @ Lam (x, t, _)     => (x,t) :: envOf (p)
    case p @ Let (x, t, _, _) => (x,t) :: envOf (p)
    case p : Exp                => envOf (p)
  }
```

Type checking: numbers, lambdas and lets

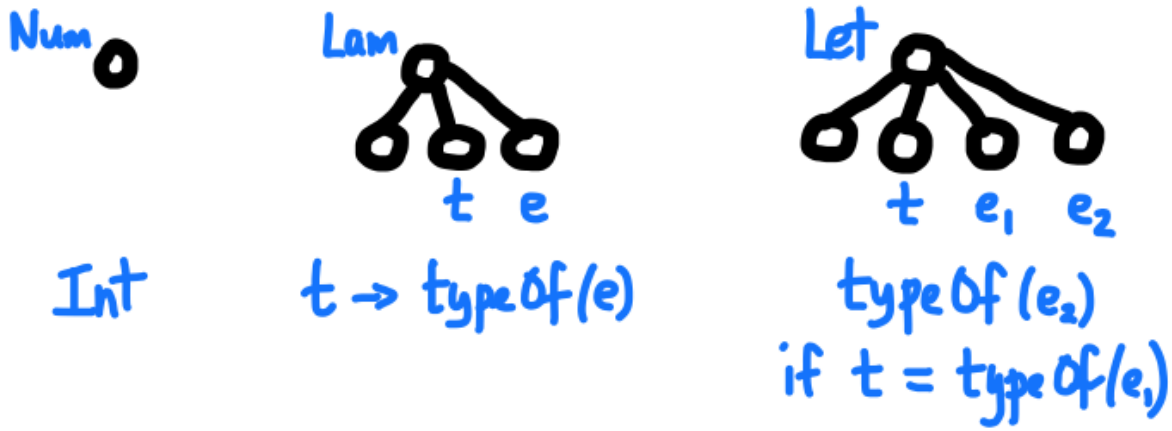

Figure : Lambdas and lets get types from their children

```
case Num (_)                => IntType ()
case Lam (_, t, e)          => FunType (t, typeOf (e))
case Let (_, t, e1, e2) => if (checkType (e1, t))
                              typeOf (e2)
                           else
                              UnknownType ()
```

Type checking: application

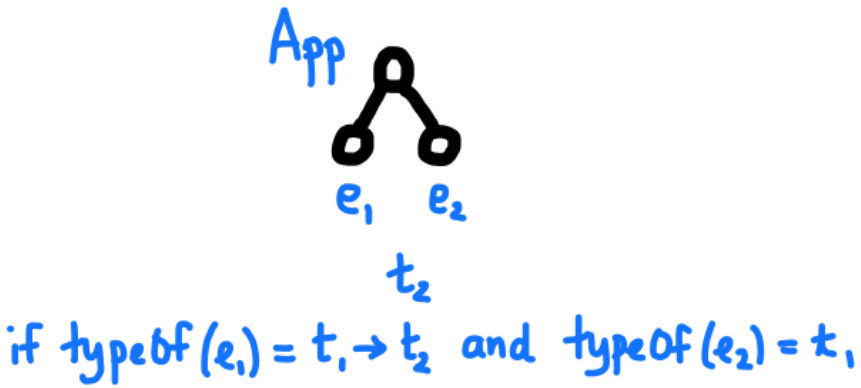

Figure : Applications get types from the applied functions

```
case App (e1, e2) =>
  typeOf (e1) match {
    case FunType (t1, t2) =>
      if (checkType (e2, t1)) t2
      else UnknownType ()
    case _ =>
      message (e1, e1 + ": non-function applied")
      UnknownType ()
  }
```

Type checking: variables

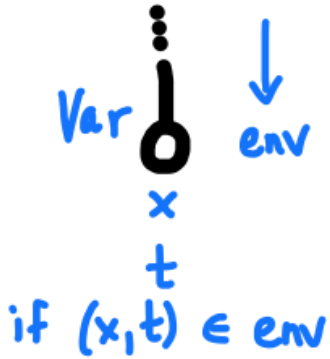

Figure : Variables get their types from the environment

```
case e @ Var (x) =>
  envOf (e).collectFirst {
    case (y, t) if x == y => t
  }.getOrElse {
    message (e, x + ": unknown variable")
    UnknownType ()
  }
```

Why should I care?

- Lambda calculus is fun!
  - bitbucket.org/inkytonik/lambdajam13
- Many problems take this form:
  - extracting information from structured data
  - checking that structured data meets constraints
  - translating structured data into other forms
  - using analysis results to guide transformation
- Abstracting away from details of traversal and storage
  - focusses your attention on the transformations and the analyses,
  - simplifies what you have to write, and
  - makes the computation more independent of the structure.

Conclusion

- Kiama abstracts language processing problems by tree rewriting and tree decoration.
- Details of traversal and storage are almost entirely implicit.
- The focus is on the transformations and analyses.
- Scala enables these DSLs to be implemented easily and they integrate naturally with other code.
- Wait there's more!
  - rewriting collections
  - pretty-printing domain-specific language
  - frameworks for compilers, REPLs and testing
  - augmentation of Scala parser combinator library
  - profiling of rewriting and attribution

More Information

- Kiama project: `kiama.googlecode.com`
  - Papers and talks: `code.google.com/p/kiama/wiki/Research`
- Other projects:
  - Using macros to get names: `bitbucket.org/inkytonik/dsinfo`
  - Domain-specific profiling: `bitbucket.org/inkytonik/dsprofile`
  - sbt plugin for Rats! parser generator: `sbt-rats.googlecode.com`
  - Scala worksheet for Sublime Text 3: `bitbucket.org/inkytonik/scalaworksheet`
- Related work:
  - Stratego: `strategoxt.org`
  - JastAdd: `jastadd.org`
  - Swierstra, S., and Chitil, O. *Linear, bounded, functional pretty printing*. Journal of Functional Programming 19, 1 (2008), 1–16

Workshop

- Lambda calculus
  - New feature:
    - matching by cases
  - New processing:
    - different evaluation strategies
    - profiling demonstration
- Kiama
  - Implementation of rewriting and attribution DSLs
  - How do we use Scala macros?
  - Pretty-printing DSL
  - How does profiling work?