Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System

Jeff Smits, Gabriël Konat, **Eelco Visser**

IFIP WG2.11 | Paris | February 19, 2020

An Incremental Compiler for Stratego using the PIE build system

Jeff Smits, Gabriël Konat, **Eelco Visser**

IFIP WG2.11 | Paris | February 1?, 2020

Background: Meta-Programming with Stratego

Building Program Optimizers with Rewriting Strategies
Eelco Visser, Zine-El-Abidine Benaissa, Andrew P. Tolmach.
ICFP 1998 [pdf, doi, bib, researchr, abstract]

Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9
Eelco Visser.
Dagstuhl 2003 [pdf, doi, bib, researchr, abstract]

Program Transformation with Scoped Dynamic Rewrite Rules
Martin Bravenboer, Arthur van Dam, Karina Olmos, Eelco Visser.
FUIN 69(1-2) 2006 [pdf, doi, bib, researchr, abstract]

Stratego/XT 0.17. A language and toolset for program transformation
Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, Eelco Visser.
SCP 72(1-2) 2008 [pdf, doi, bib, researchr, abstract]

The Spoofax language workbench: rules for declarative specification of languages and IDEs
Lennart C. L. Kats, Eelco Visser.
OOPSLA 2010 [pdf, doi, bib, researchr, abstract]

```
rules

[Ri]
   [A.C1 = [A.C2 = t11 op2 t12] op1 t2] ⟶ [A.C2 = t11 op2 [A.C1 = t12 op1 t2]]
   where [A.C1 = [A.C2 = A op2 A] op1 A]

[Le]
   [A.C1 = t1 op1 [A.C2 = t21 op2 t22]] ⟶ [A.C2 = [A.C1 = t1 op1 t21] op2 t22]
   where [A.C1 = A op1 [A.C2 = A op2 A]]

critical pairs

([Ri]+[Ri])
  [A.C1 = [A.C2 = [A.C3 = t32 op32 t33] op22 t22] op12 t12]

  conflict patterns
    {[A.C1 = [A.C2 = A op22 A] op12 A],
     [A.C2 = [A.C3 = A op32 A] op22 A]}

  derived relations
    {(C1 > C2 and C2 > C3 ⟹ C1 > C3)
    or (C1 > C2 and C2 right C3 ⟹ C1 > C3)
    or (C1 right C2 and C2 > C3 ⟹ C1 > C3)
    or (C1 right C2 and C2 right C3 ⟹ C1 right C3)}

  derived conflict patterns
    {[A.C1 = [A.C3 = A op32 A] op12 A],
     [A.C1 = A op12 [A.C3 = A op32 A]]};
    {[A.C1 = [A.C3 = A op32 A] op12 A],
     [A.C3 = [A.C1 = A op12 A] op32 A]}

  derivations

    using {[A.C1 = [A.C3 = A op32 A] op12 A],
           [A.C1 = A op12 [A.C3 = A op32 A]]}
    left
      |  [A.C1 = [A.C2 = [A.C3 = t32 op32 t33] op22 t22] op12 t12]
      ⟶ [A.C2 = [A.C3 = t32 op32 t33] op22 [A.C1 = t22 op12 t12]] (Ri: [A.C1 = [A.C2 = A op22 A] op12 A])
      ⟶ [A.C3 = t32 op32 [A.C2 = t33 op22 [A.C1 = t22 op12 t12]]] (Ri: [A.C2 = [A.C3 = A op32 A] op22 A])
    right
      |  [A.C1 = [A.C2 = [A.C3 = t32 op32 t33] op22 t22] op12 t12]
      ⟶ [A.C1 = [A.C3 = t32 op32 [A.C2 = t33 op22 t22]] op12 t12] (Ri: [A.C2 = [A.C3 = A op32 A] op22 A])
      ⟶ [A.C3 = t32 op32 [A.C1 = [A.C2 = t33 op22 t22] op12 t12]] (Ri: [A.C1 = [A.C3 = A op32 A] op12 A])
      ⟶ [A.C3 = t32 op32 [A.C2 = t33 op22 [A.C1 = t22 op12 t12]]] (Ri: [A.C1 = [A.C2 = A op22 A] op12 A])
    confluent
      true
```

```
context-free syntax

  Rule.Rule = <
    [<LABEL>]
       <Tree> -\> <Tree>
       where <Condition>
  >

  Rule.Rule = <
    [<LABEL>]
       <Tree> -\> <Tree>
  >

  Condition.Rel = Rel
  Condition.Tree = Tree

  Conditions.Conditions = <
    {<{Condition “,\n”}*>}
  >

context-free syntax // trees

  Tree.Node = <
    [<NonTerm>.<Cns> = <Tree*>]
  >

  Tree.Var = <<VAR>>
```

```
rules

  unify =
    repeat(Unify)

  Unify :
    ([(Var(x), t) | pair1*], subs1) → (pair2*, [(Var(x), t) | subs2])
    with not(<oncetd(?Var(x); debug(!"  - occurs check fails: "))> t)
    with <subst(|[(Var(x), t)])> (pair1*, subs1) ⇒ (pair2*, subs2)

  Unify :
    ([(t, Var(x)) | pair1*], subs1) → ([(Var(x), t) | pair1*], subs1)

  Unify :
    ([(Node(nt1, c1, ts1), Node(nt2, c2, ts2)) | pair1*], subs) →
     ([(nt1, nt2), (c1, c2), pair2*, pair1*], subs)
    where <zip>(ts1, ts2) ⇒ pair2*
```

```
rules

  rel-to-conclusions =
    conclusions
    ; innermost(IdR <+ AndT <+ OrF <+ OrSw <+ OrL)

  dnf = innermost(DM + OrL + OrF + AndT + AndImpl + AndNot)

  DM : And(Or(a, b), c) → Or(And(a, c), And(b, c))
  DM : And(a, Or(b, c)) → Or(And(a, b), And(a, c))

  OrL : Or(a, Or(b, c)) → Or(Or(a, b), c)
  AndL : And(a, And(b, c)) → And(And(a, b), c)

  OrF : Or(a, False()) → a
  OrF : Or(False(), a) → a

  OrT : Or(True(), a) → True()
  OrT : Or(a, True()) → True()

  AndT : And(True(), a) → a
  AndT : And(a, True()) → a

  OrSw : Or(a@Left(_,_), b@Prio(_,_)) → Or(b, a)
  OrSw : Or(a@Right(_,_), b@Prio(_,_)) → Or(b, a)

  OrSw : Or(Or(a, b@Left(_,_)), c@Prio(_,_)) → Or(Or(a, c), b)
  OrSw : Or(Or(a, b@Right(_,_)), c@Prio(_,_)) → Or(Or(a, c), b)
```

Some Stratego Properties

Directly operating on (abstract) syntax

- Terms are abstract syntax trees; correspond directly to parse trees
- Parser produces terms; pretty-printer produces text

Meta-programming with concrete object syntax

Term rewriting

- Define transformations as (labeled) term rewrite rules

Rewriting strategies

- Control the application of rewrite rules

Cross-module extensibility of rules and strategies

```
module lang/booleans/syntax

context-free syntax

  Type.BoolT = <Bool>

  Exp.Eq    = <<Exp> == <Exp>> {non-assoc}

  Exp.True  = <true>
  Exp.False = <false>
  Exp.Not   = <!<Exp>>
  Exp.And   = <<Exp> && <Exp>> {left}
  Exp.Or    = <<Exp> || <Exp>> {left}

  Exp.If = <
    if <Exp> then
      <Exp>
    else
      <Exp>
  >

  Exp.IfT = <
    if <Exp> then
      <Exp>
  >

  Val.BoolV = <<BoolV>>
  BoolV.True = <true>
  BoolV.False = <false>

context-free priorities

  Exp.Not > Exp.Eq > Exp.And
  > Exp.Or > Exp.IfT > Exp.If
```

```
module lang/booleans/statics

imports lang/base/statics
imports lang/unit/statics

rules

  typeOfExp(s, True()) = BOOL().

  typeOfExp(s, False()) = BOOL().

  typeOfExp(s, Not(e)) = BOOL() :-
    typeOfExp(s, e) == BOOL().

  typeOfExp(s, And(e1, e2)) = BOOL() :-
    typeOfExp(s, e1) == BOOL(),
    typeOfExp(s, e2) == BOOL().

  typeOfExp(s, Or(e1, e2)) = BOOL() :-
    typeOfExp(s, e1) == BOOL(),
    typeOfExp(s, e2) == BOOL().

  typeOfExp(s, If(e1, e2, e3)) = lub(T1, T2) :-
    typeOfExp(s, e1) == BOOL(),
    typeOfExp(s, e2) == T1,
    typeOfExp(s, e3) == T2,
    equitype(T1, T2).

  typeOfExp(s, IfT(e1, e2)) = UNIT() :- {T}
    typeOfExp(s, e1) == BOOL(),
    typeOfExp(s, e2) == T.

  typeOfExp(s, Eq(e1, e2)) = BOOL() :- {T1 T2}
    typeOfExp(s, e1) == T1,
    typeOfExp(s, e2) == T2,
    equitype(T1, T2).
```

```
module lang/booleans/dynamics

imports signatures/lang/base/syntax-sig
imports signatures/lang/booleans/syntax-sig

rules

  eval(|f) :
    True() --> BoolV(True())

  eval(|f) :
    False() --> BoolV(False())

  eval(|f) :
    Not(e) --> <notb>v
    with <eval(|f)> e => v

  eval(|f) :
    And(e1, e2) --> <andb(|f)>(v1, e2)
    with <eval(|f)> e1 => v1
    with <eval(|f)> e2 => v2

  eval(|f) :
    Or(e1, e2) --> <orb(|f)>(v1, v2)
    with <eval(|f)> e1 => v1
    with <eval(|f)> e2 => v2

  eval(|f) :
    If(e1, e2, e3) --> v
    with <eval(|f)> e1 => v1
    with <ifb(|f)> (v1, e2, e3) => v

  eval(|f) :
    Eq(e1, e2) --> v
    with <eval(|f)> e1 => v1
    with <eval(|f)> e2 => v2
    with <eqb> (e1, e2) => v
```

```
module lang/booleans/syntax

context-free syntax

  Type.BoolT = <Bool>

  Exp.Eq    = <<Exp> == <Exp>> {non-assoc}

  Exp.True  = <true>
  Exp.False = <false>
  Exp.Not   = <!<Exp>>
  Exp.And   = <<Exp> && <Exp>> {left}
  Exp.Or    = <<Exp> || <Exp>> {left}

  Exp.If = <
    if <Exp> then
      <Exp>
    else
      <Exp>
  >

  Exp.IfT = <
    if <Exp> then
      <Exp>
  >

  Val.BoolV = <<BoolV>>
  BoolV.True = <true>
  BoolV.False = <false>

context-free priorities

  Exp.Not > Exp.Eq > Exp.And
  > Exp.Or > Exp.IfT > Exp.If
```

```
module lang/booleans/dynamics

imports signatures/lang/base/syntax-sig
imports signatures/lang/booleans/syntax-sig

rules

  eval(|f) :
    True() → BoolV(True())

  eval(|f) :
    False() → BoolV(False())

  eval(|f) :
    Not(e) → <notb>v
    with <eval(|f)> e ⇒ v

  eval(|f) :
    And(e1, e2) → <andb(|f)>(v1, e2)
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2

  eval(|f) :
    Or(e1, e2) → <orb(|f)>(v1, v2)
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2

  eval(|f) :
    If(e1, e2, e3) → v
    with <eval(|f)> e1 ⇒ v1
    with <ifb(|f)> (v1, e2, e3) ⇒ v

  eval(|f) :
    Eq(e1, e2) → v
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2
    with <eqb> (e1, e2) ⇒ v
```

```
module lang/booleans/syntax

context-free syntax

  Type.BoolT = <Bool>

  Exp.Eq    = <<Exp> == <Exp>> {non-assoc}

  Exp.True  = <true>
  Exp.False = <false>
  Exp.Not   = <!<Exp>>
  Exp.And   = <<Exp> && <Exp>> {left}
  Exp.Or    = <<Exp> || <Exp>> {left}

  Exp.If = <
    if <Exp> then
      <Exp>
    else
      <Exp>
  >

  Exp.IfT = <
    if <Exp> then
      <Exp>
  >

  Val.BoolV = <<BoolV>>
  BoolV.True = <true>
  BoolV.False = <false>

context-free priorities

  Exp.Not > Exp.Eq > Exp.And
  > Exp.Or > Exp.IfT > Exp.If
```

```
module lang/booleans/dynamics

imports signatures/lang/base/syntax-sig
imports signatures/lang/booleans/syntax-sig

rules

  eval(|f) :
    True() → BoolV(True())

  eval(|f) :
    False() → BoolV(False())

  eval(|f) :
    Not(e) → <notb>v
    with <eval(|f)> e ⇒ v

  eval(|f) :
    And(e1, e2) → <andb(|f)>(v1, e2)
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2

  eval(|f) :
    Or(e1, e2) → <orb(|f)>(v1, v2)
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2

  eval(|f) :
    If(e1, e2, e3) → v
    with <eval(|f)> e1 ⇒ v1
    with <ifb(|f)> (v1, e2, e3) ⇒ v

  eval(|f) :
    Eq(e1, e2) → v
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2
    with <eqb> (e1, e2) ⇒ v
```

```
module lang/arithmetic/syntax

imports lang/base/syntax

context-free syntax // arithmetic

  Type.IntT = <Int>

  Exp.Int   = <<INT>>
  Exp.Add   = <<Exp> + <Exp>> {left}
  Exp.Sub   = <<Exp> - <Exp>> {left}
  Exp.Mul   = <<Exp> * <Exp>> {left}

  Val.IntV  = <<INT>>

context-free priorities

  Exp.Mul > {left: Exp.Add Exp.Sub}
```

```
module lang/arithmetic/dynamics

imports signatures/lang/base/syntax-sig
imports signatures/lang/arithmetic/syntax-sig

imports lang/booleans/dynamics

signature
  constructors
    IntV : Int → Val

rules

  eval(|f) :
    Int(i) → IntV(i)

  eval(|f) :
    Add(e1, e2) → IntV(<addS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)

  eval(|f) :
    Sub(e1, e2) → IntV(<subtS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)

  eval(|f) :
    Mul(e1, e2) → IntV(<mulS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)
```

- > lang
  - > arithmetic
  - > base
  - > booleans
  - > file
  - function
  - > generics
  - > L1
  - module
  - record
  - string
  - type
  - union
  - unit
  - variable

- > lang
  - > arithmetic
    - dynamics.str
    - > statics.stx
    - > syntax.sdf3
  - > base
    - dynamics.str
    - frames.str
    - lexical.sdf3
    - > statics.stx
    - > syntax.sdf3
  - > booleans
    - dynamics.str
    - > statics.stx
    - > syntax.sdf3
  - > file
    - dynamics.str
    - > statics.stx
    - > syntax.sdf3
  - function
    - statics.stx
    - syntax.sdf3
  - > generics
    - statics.stx
    - syntax.sdf3
  - > L1
    - dynamics.str
    - > statics.stx
    - > syntax.sdf3
  - module
    - statics.stx
    - syntax.sdf3
  - record
    - statics.stx
    - syntax.sdf3
  - string
    - statics.stx
    - syntax.sdf3
  - type
    - statics.stx
    - syntax.sdf3
  - union
    - statics.stx
    - syntax.sdf3
  - unit
    - statics.stx
    - syntax.sdf3
  - variable
    - statics.stx
    - syntax.sdf3

```
module lang/booleans/syntax

context-free syntax

  Type.BoolT = <Bool>

  Exp.Eq    = <<Exp> == <Exp>> {non-assoc}

  Exp.True  = <true>
  Exp.False = <false>
  Exp.Not   = <!<Exp>>
  Exp.And   = <<Exp> && <Exp>> {left}
  Exp.Or    = <<Exp> || <Exp>> {left}

  Exp.If = <
    if <Exp> then
      <Exp>
    else
      <Exp>
  >

  Exp.IfT = <
    if <Exp> then
      <Exp>
  >

  Val.BoolV = <<BoolV>>
  BoolV.True = <true>
  BoolV.False = <false>

context-free priorities

  Exp.Not > Exp.Eq > Exp.And
  > Exp.Or > Exp.IfT > Exp.If
```

```
module lang/booleans/dynamics

imports signatures/lang/base/syntax-sig
imports signatures/lang/booleans/syntax-sig

rules

  eval(|f) :
    True() → BoolV(True())

  eval(|f) :
    False() → BoolV(False())

  eval(|f) :
    Not(e) → <notb>v
    with <eval(|f)> e ⇒ v

  eval(|f) :
    And(e1, e2) → <andb(|f)>(v1, e2)
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2

  eval(|f) :
    Or(e1, e2) → <orb(|f)>(v1, v2)
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2

  eval(|f) :
    If(e1, e2, e3) → v
    with <eval(|f)> e1 ⇒ v1
    with <ifb(|f)> (v1, e2, e3) ⇒ v

  eval(|f) :
    Eq(e1, e2) → v
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2
    with <eqb> (e1, e2) ⇒ v
```

```
module lang/arithmetic/syntax

imports lang/base/syntax

context-free syntax // arithmetic

  Type.IntT = <Int>

  Exp.Int   = <<INT>>
  Exp.Add   = <<Exp> + <Exp>> {left}
  Exp.Sub   = <<Exp> - <Exp>> {left}
  Exp.Mul   = <<Exp> * <Exp>> {left}

  Val.IntV  = <<INT>>

context-free priorities

  Exp.Mul > {left: Exp.Add Exp.Sub}
```

```
module lang/arithmetic/dynamics

imports signatures/lang/base/syntax-sig
imports signatures/lang/arithmetic/syntax-sig

imports lang/booleans/dynamics

signature
  constructors
    IntV : Int → Val

rules

  eval(|f) :
    Int(i) → IntV(i)

  eval(|f) :
    Add(e1, e2) → IntV(<addS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)

  eval(|f) :
    Sub(e1, e2) → IntV(<subtS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)

  eval(|f) :
    Mul(e1, e2) → IntV(<mulS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)
```

```
module lang/variable/syntax

imports lang/base/syntax

context-free syntax // variables

  Exp.Var = <<ID>>

  Decl.Def = <def <Bind>>

  Bind.Bind = <<ID> = <Exp>>

  Bind.BindTyped = <<ID> : <Type> = <Exp>>

  Exp.LetSeq = <
    let
      <{Bind ";\n"}*>
    in
      <Exp>
  >
  Exp.LetPar = <
    letpar
      <{Bind ";\n"}*>
    in
      <Exp>
  >
  Exp.LetRec = <
    letrec
      <{Bind ";\n"}*>
    in
      <Exp>
  >
```

```
module lang/record/syntax

imports lang/base/syntax

context-free syntax // record types

  Type.RecT = <{<{FDecl ",\n"}*>}>

  Decl.Record = <
    record <ID> {
      <{FDecl ",\n"}*>
    }
  >

  FDecl.FDecl = <<ID> : <Type>>

context-free syntax // record expressions

  Exp.New = <new <ID> { <{FBind ","}*> }>

  Exp.Cns = <<ID> { <{FBind ","}*> }>

  FBind.FBind = <<ID> = <Exp>>

  Exp.Proj = <<Exp>.<ID>>

  Exp.With = <
    with <Exp> do
      <Exp>
  >

  Exp.Null = <null>
```

Compiling Stratego

(Compiling cross-module extensibility)

```
module lang/booleans/syntax

context-free syntax

  Type.BoolT = <Bool>

  Exp.Eq    = <<Exp> == <Exp>> {non-assoc}

  Exp.True  = <true>
  Exp.False = <false>
  Exp.Not   = <!<Exp>>
  Exp.And   = <<Exp> && <Exp>> {left}
  Exp.Or    = <<Exp> || <Exp>> {left}

  Exp.If = <
    if <Exp> then
      <Exp>
    else
      <Exp>
  >

  Exp.IfT = <
    if <Exp> then
      <Exp>
  >

  Val.BoolV = <<BoolV>>
  BoolV.True = <true>
  BoolV.False = <false>

context-free priorities

  Exp.Not > Exp.Eq > Exp.And
  > Exp.Or > Exp.IfT > Exp.If
```

```
module lang/booleans/dynamics

imports signatures/lang/base/syntax-sig
imports signatures/lang/booleans/syntax-sig

rules

  eval(|f) :
    True() → BoolV(True())

  eval(|f) :
    False() → BoolV(False())

  eval(|f) :
    Not(e) → <notb>v
    with <eval(|f)> e ⇒ v

  eval(|f) :
    And(e1, e2) → <andb(|f)>(v1, e2)
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2

  eval(|f) :
    Or(e1, e2) → <orb(|f)>(v1, v2)
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2

  eval(|f) :
    If(e1, e2, e3) → v
    with <eval(|f)> e1 ⇒ v1
    with <ifb(|f)> (v1, e2, e3) ⇒ v

  eval(|f) :
    Eq(e1, e2) → v
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2
    with <eqb> (e1, e2) ⇒ v
```

```
module lang/arithmetic/syntax

imports lang/base/syntax

context-free syntax // arithmetic

  Type.IntT = <Int>

  Exp.Int   = <<INT>>
  Exp.Add   = <<Exp> + <Exp>> {left}
  Exp.Sub   = <<Exp> - <Exp>> {left}
  Exp.Mul   = <<Exp> * <Exp>> {left}

  Val.IntV  = <<INT>>

context-free priorities

  Exp.Mul > {left: Exp.Add Exp.Sub}
```

```
module lang/arithmetic/dynamics

imports signatures/lang/base/syntax-sig
imports signatures/lang/arithmetic/syntax-sig

imports lang/booleans/dynamics

signature
  constructors
    IntV : Int → Val

rules

  eval(|f) :
    Int(i) → IntV(i)

  eval(|f) :
    Add(e1, e2) → IntV(<addS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)

  eval(|f) :
    Sub(e1, e2) → IntV(<subtS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)

  eval(|f) :
    Mul(e1, e2) → IntV(<mulS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)
```

```
module lang/booleans/dynamics

imports signatures/lang/base/syntax-sig
imports signatures/lang/booleans/syntax-sig

rules

  eval(|f) :
    True() --> BoolV(True())

  eval(|f) :
    False() --> BoolV(False())

  eval(|f) :
    Not(e) --> <notb>v
    with <eval(|f)> e => v

  eval(|f) :
    And(e1, e2) --> <andb(|f)>(v1, e2)
    with <eval(|f)> e1 => v1
    with <eval(|f)> e2 => v2

  eval(|f) :
    Or(e1, e2) --> <orb(|f)>(v1, v2)
    with <eval(|f)> e1 => v1
    with <eval(|f)> e2 => v2

  eval(|f) :
    If(e1, e2, e3) --> v
    with <eval(|f)> e1 => v1
    with <ifb(|f)> (v1, e2, e3) => v

  eval(|f) :
    Eq(e1, e2) --> v
    with <eval(|f)> e1 => v1
    with <eval(|f)> e2 => v2
    with <eqb> (e1, e2) => v
```

```
module lang/arithmetic/dynamics

imports signatures/lang/base/syntax-sig
imports signatures/lang/arithmetic/syntax-sig

imports lang/booleans/dynamics

signature
  constructors
    IntV : Int -> Val

rules

  eval(|f) :
    Int(i) --> IntV(i)

  eval(|f) :
    Add(e1, e2) --> IntV(<addS>(i, j))
    with <eval(|f)> e1 => IntV(i)
    with <eval(|f)> e2 => IntV(j)

  eval(|f) :
    Sub(e1, e2) --> IntV(<subtS>(i, j))
    with <eval(|f)> e1 => IntV(i)
    with <eval(|f)> e2 => IntV(j)

  eval(|f) :
    Mul(e1, e2) --> IntV(<mulS>(i, j))
    with <eval(|f)> e1 => IntV(i)
    with <eval(|f)> e2 => IntV(j)
```

```
module lang/booleans/dynamics


rules

  eval(|f) :
    Not(e) → <notb>v
    with <eval(|f)> e ⇒ v
```

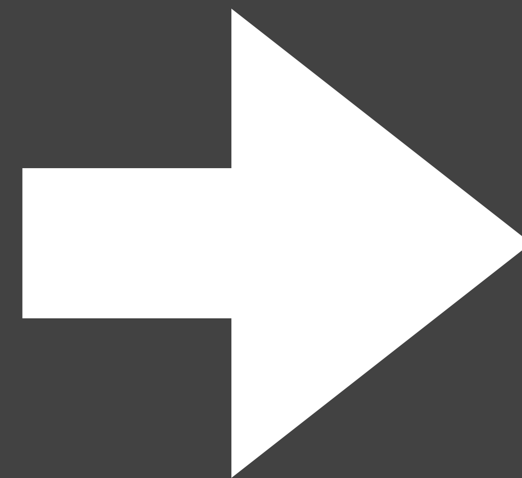
```
module lang/arithmetic/dynamics

rules

  eval(|f) :
    Add(e1, e2) → IntV(<addS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)
```

```
module lang/booleans/dynamics


rules

  eval(|f) :
    Not(e) → <notb>v
    with <eval(|f)> e ⇒ v
```

```
module lang/arithmetic/dynamics

rules

  eval(|f) :
    Add(e1, e2) → IntV(<addS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)
```

```
module lang/booleans/dynamics


rules

  eval(|f) :
    Not(e) → <notb>v
    with <eval(|f)> e ⇒ v
```

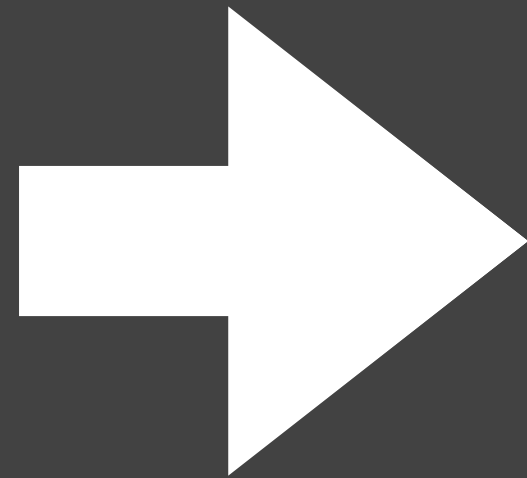
```
module lang/arithmetic/dynamics

rules

  eval(|f) :
    Add(e1, e2) → IntV(<addS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)
```



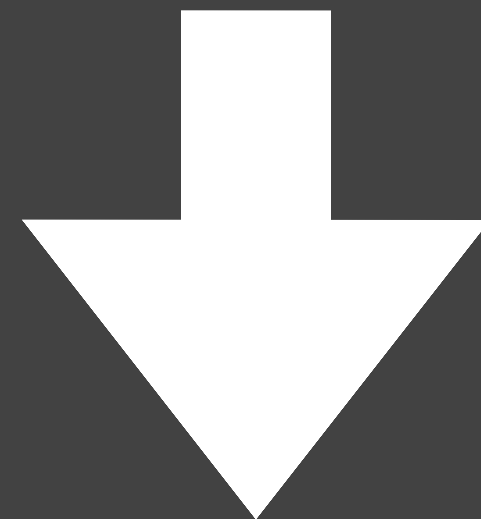
```
strategies

  eval(|f) = {e, v
             : ?Not(e)
             ; with(<eval(|f)> e ⇒ v)
             ; <notb>v}
          <+ {e1, e2, i, j
             : ?Add(e1, e2)
             ; with(<eval(|f)> e1 ⇒ IntV(i))
             ; with(<eval(|f)> e2 ⇒ IntV(j))
             ; !IntV(<addS>(i, j))}
```

```
 v
```

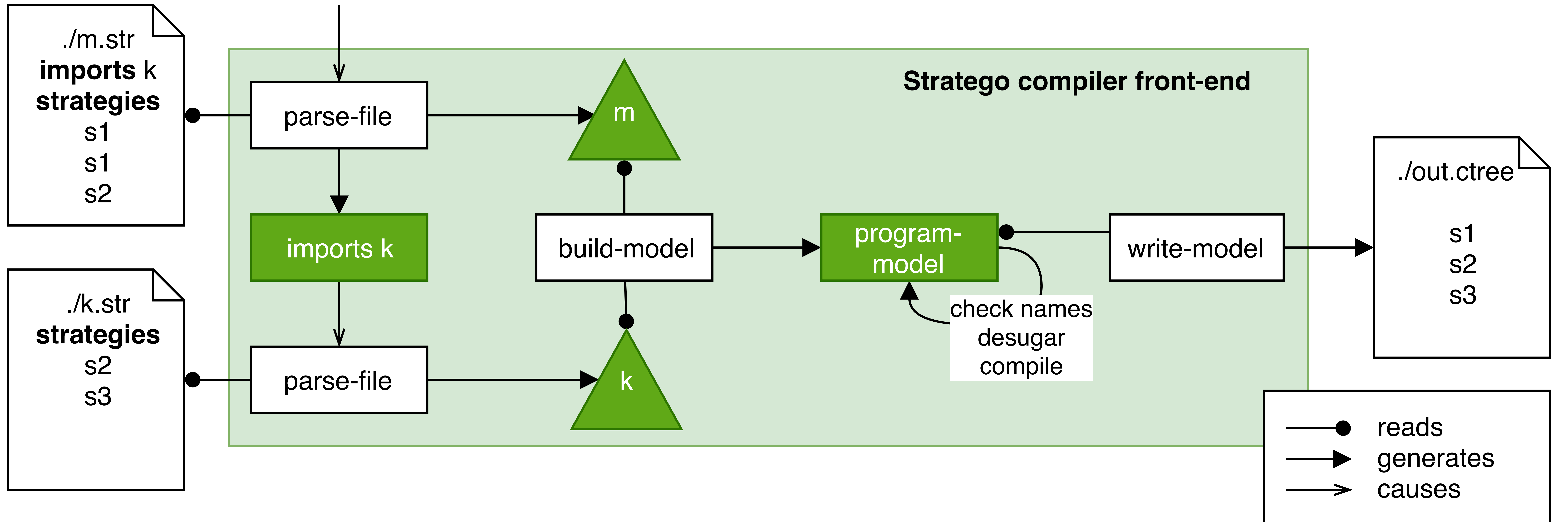
```
ynamics



<addS>(i, j))
> IntV(i)
> IntV(j)
```

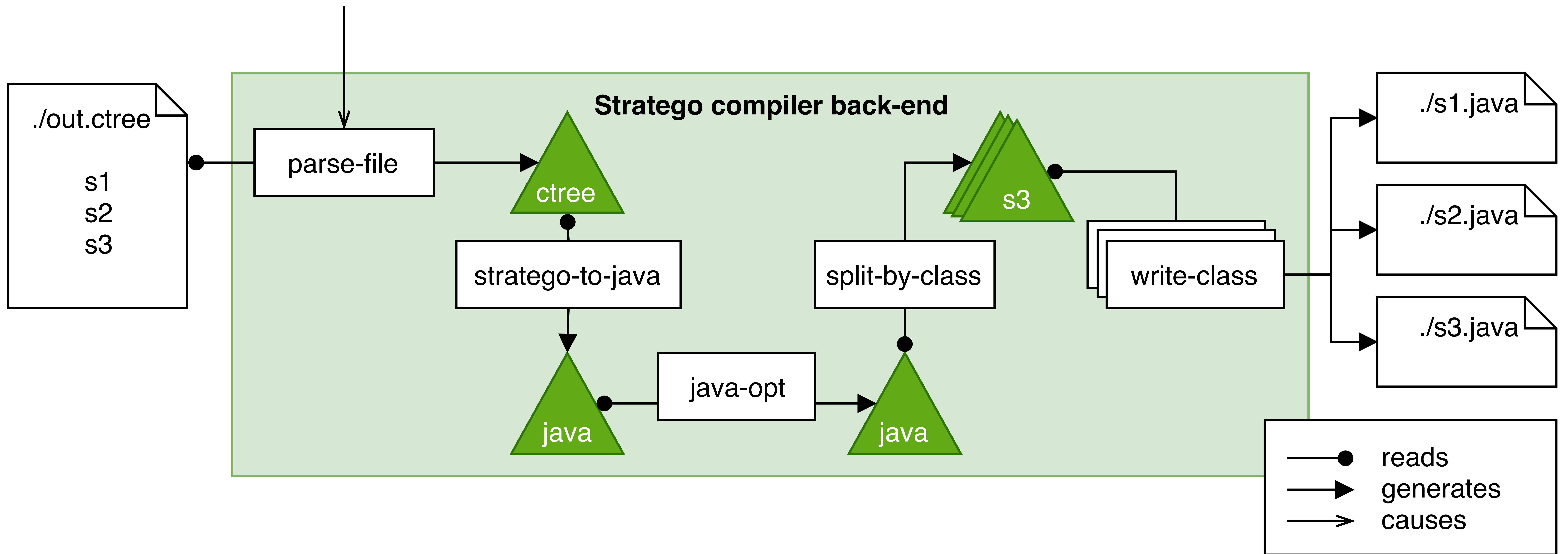


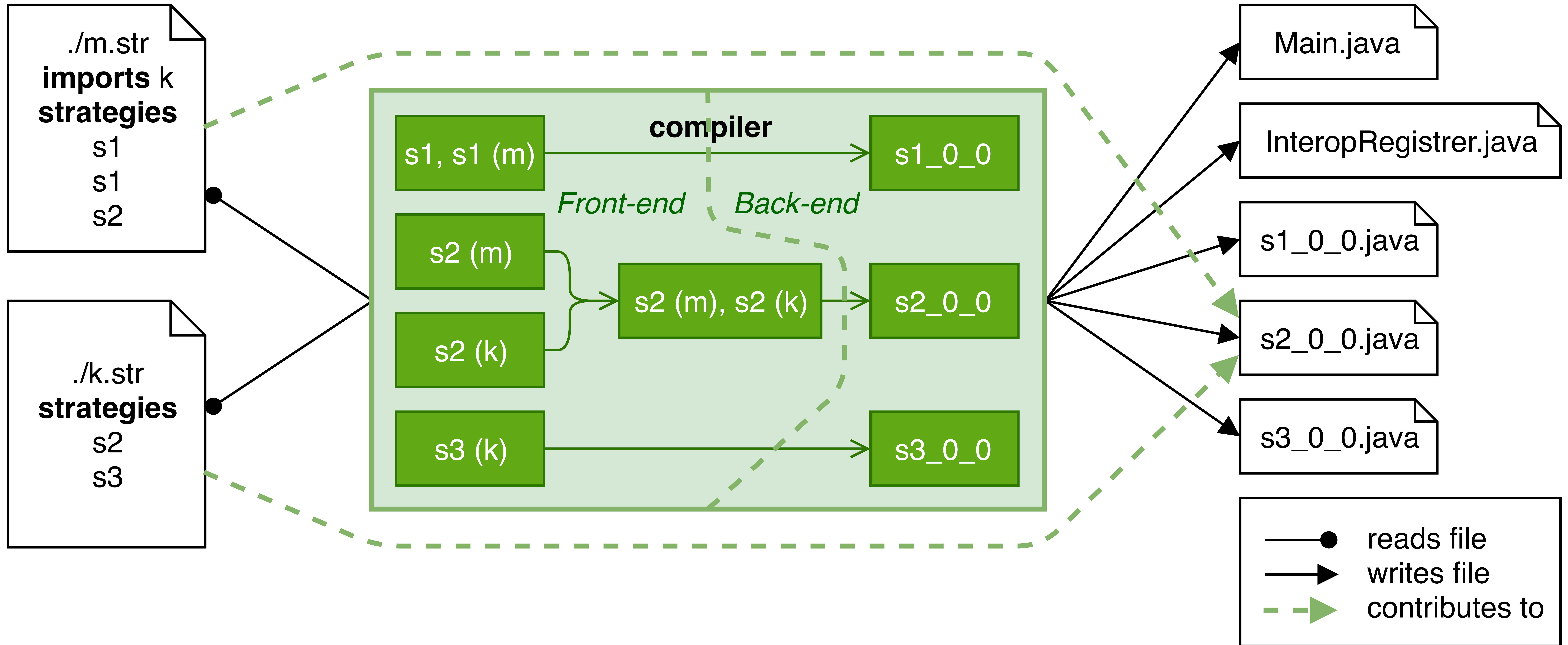
```
strategies

  eval(|f) = {e, v
               : ?Not(e)
               ; with(<eval(|f)> e ⇒ v)
               ; <notb>v}
          <+ {e1, e2, i, j
               : ?Add(e1, e2)
               ; with(<eval(|f)> e1 ⇒ IntV(i))
               ; with(<eval(|f)> e2 ⇒ IntV(j))
               ; !IntV(<addS>(i, j))}
```



```java
public class eval_0_1 extends Strategy {

  public static Strategy instance = new eval_0_1();

  public IStrategoTerm invokeDynamic(IStrategoTerm term, Stratego[] sargs, IStrategoTerm[] targs) {
    // try Not rule
      term = notb_0_0.instance.invoke(term);
      // try Add rule
      // ...
      return term;
  }
}
```


./m.str
imports k
strategies
s1
s1
s2
./k.str
strategies
s2
s3
Stratego compiler front-end
parse-file
imports k
parse-file
m
k
build-model
program-model
check names
desugar
compile
write-model
./out.ctree
s1
s2
s3
reads
generates
causes


Stratego compiler back-end
./out.ctree
s1
s2
s3
parse-file
ctree
stratego-to-java
java
java-opt
java
split-by-class
s3
write-class
./s1.java
./s2.java
./s3.java
reads
generates
causes


./m.str
imports k
strategies
s1
s1
s2
./k.str
strategies
s2
s3
compiler
s1, s1 (m)
s2 (m)
s2 (k)
s3 (k)
s2 (m), s2 (k)
Front-end
Back-end
s1_0_0
s2_0_0
s3_0_0
Main.java
InteropRegistrer.java
s1_0_0.java
s2_0_0.java
s3_0_0.java
reads file
writes file
contributes to

Analysis of Stratego Compilation

Whole program compilation is convenient

- The compiler is the build system!*

Whole program compilation is required

- Linking rules/strategies from multiple modules!
- Provides opportunities for optimization!

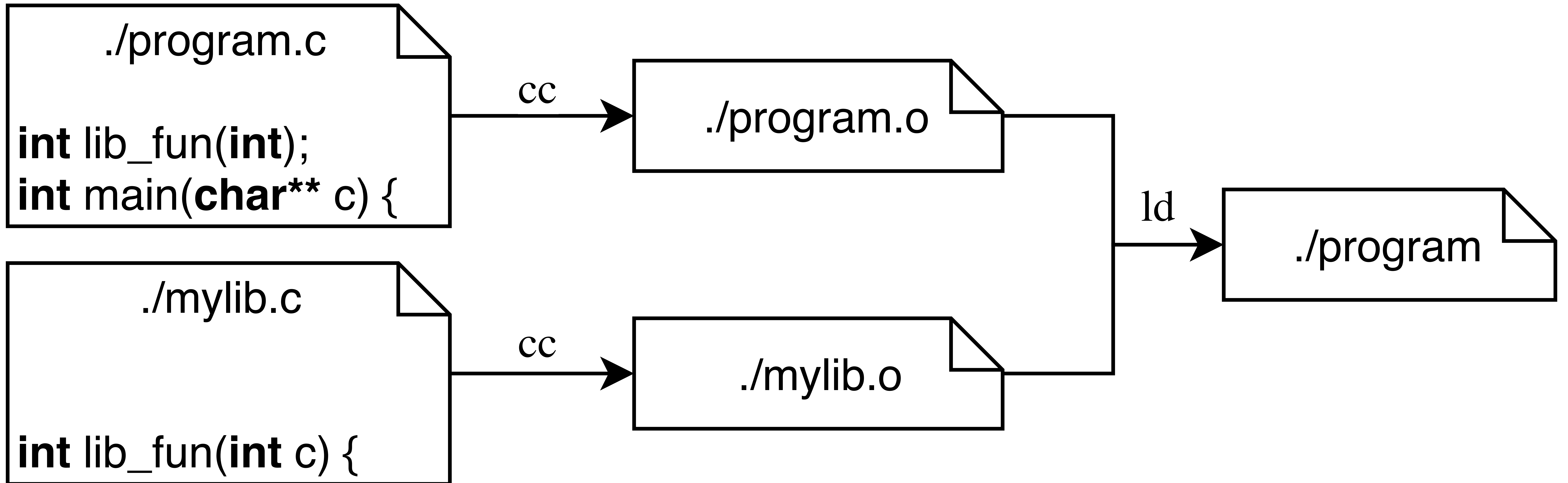
Whole program compilation is expensive

- Compile all modules for any change!

(*) Stratego has a notion of libraries; but cross-module extensibility does not apply to such libraries.

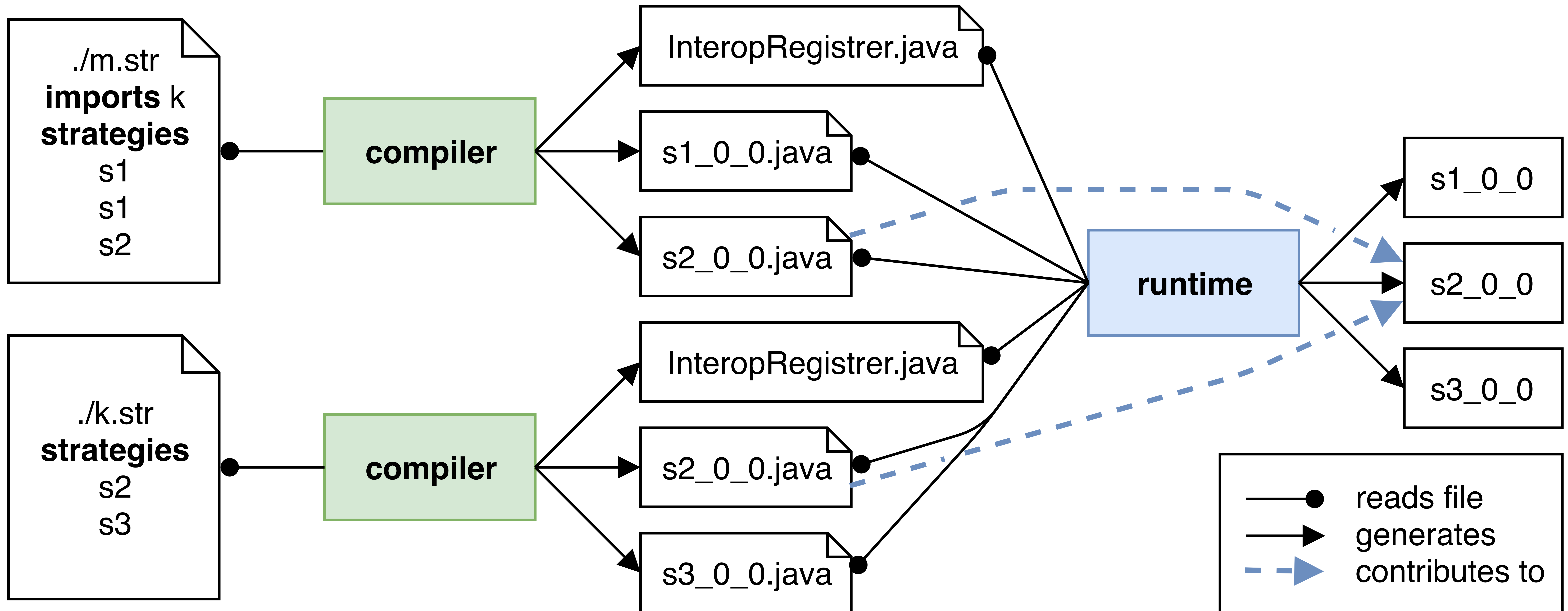
Separately Compiling Stratego Modules

Separate Compilation of C Programs



Works because function is implemented in one compilation unit

Case study for the pluto build system



A Sound and Optimal Incremental Build System with Dynamic Dependencies.
Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. OOPSLA'15.

```
strategies

  eval(|f) = {e, v
             : ?Not(e)
             ; with(<eval(|f)> e => v)
             ; <notb>v}
```

```java
public class eval_0_1 extends RegisteringStrategy {

  public static final Strategy instance = new eval_0_1();

  public static Strategy getStrategy(Context context) {
    return context.getStrategyCollector().getStrategyExecutor("eval_0_1");
  }

  public void registerImplementators(StrategyCollector collector) {
   collector.registerStrategyImplementator("eval_0_1", instance);
  }

  public void bindExecutors(StrategyCollector collector) { ... }

  public IStrategoTerm invokeDynamic(IStrategoTerm term, IStrategoTerm arg1) {
     // apply the Not rule
     term = Main.notb_0_0.invoke(term);
     // note the difference in invocation!
     return term;
  }
}
```

Analysis of Dynamic Linking Model

Requires invasive changes to the compiler

- Compilation of dynamic rules requires generation of per program rules; dynamic linking model repeated these for each module
- Never produced a version that covered entire language

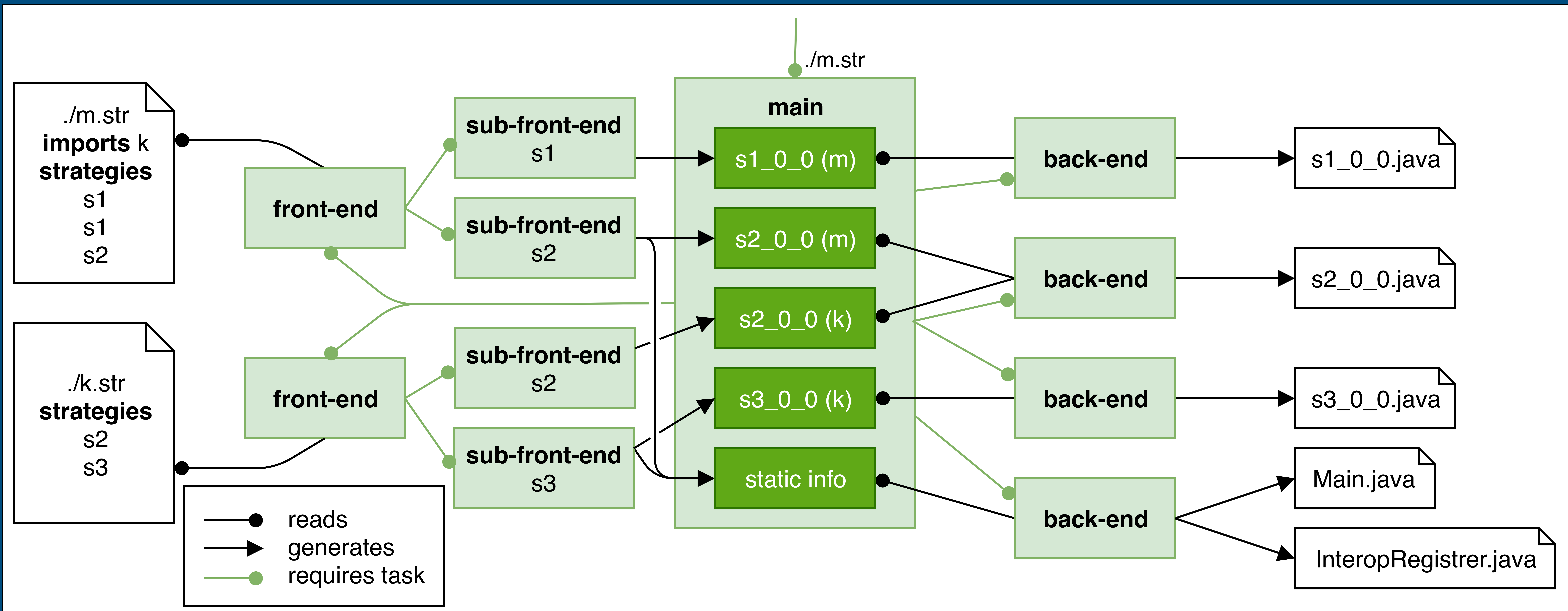
Run-time overhead

- Linking happens at run-time
- No cross-module optimizations
  - (but there may not be a lot of that in practice)
- No benchmarks available of build-time gains vs run-time overhead

Requires external build system

- Complicates the compilation process

Incrementally Compiling Stratego Modules



Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System

Jeff Smits, Gabriël Konat, Eelco Visser. To appear in <Programming> 2020

Analysis of Incremental Compilation Model

No run-time overhead, compatibility problems

- Compilation result is the same as whole program compiler

Reuse components of existing compiler

- Architecture is almost the same as original compiler
- Expose smaller units of computation for caching
- Link later in order to cache work earlier in the pipeline
- Sub-module incrementality
  - (instead of per module separate compilation)

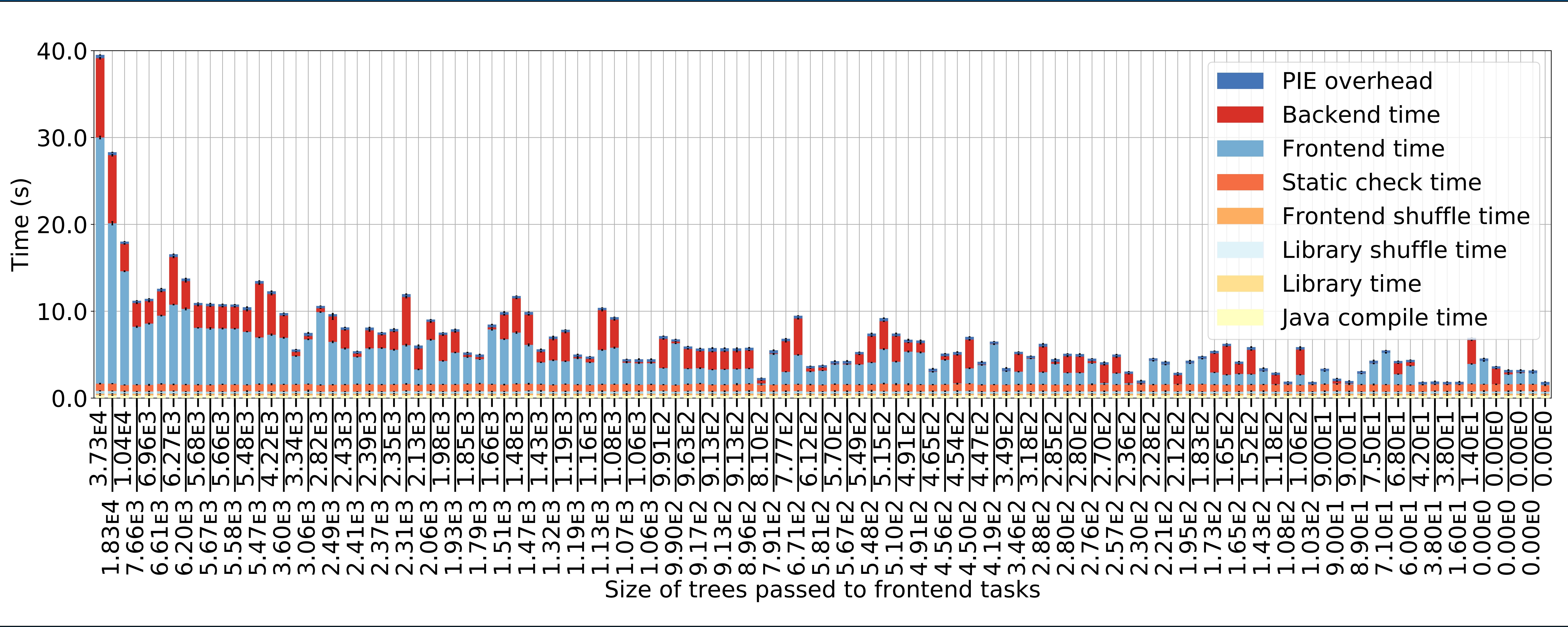
Hides build system

- Compilation process is the same as original compiler

Performance

- Speed-up of re-compilation 2x - 90x; overhead for initial compilation 2x

From 93s to …



Incrementally compiling 200 commits of the WebDSL compiler, ordered by size of change

Incremental Software Pipelines with PIE

PIE Satisfies Build System Requirements

Efficient

- Low overhead
- Rebuild time proportional to effect of change

Precise

- Yield same result as clean build (Correct)
- Re-execute minimum number of build tasks

Expressive

- Minimise accidental complexity

Pluto: Top-Down Incremental Build Algorithm

[1] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. “A sound and optimal incremental build system with dynamic dependencies”. OOPSLA 2015. DOI: 10.1145/2814270.2814316

PIE Language

[2] Gabriël Konat, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. “PIE: A Domain-Specific Language for Interactive Software Development Pipelines”. Programming 2018. DOI: 10.22152/programming-journal.org/2018/2/9

PIE Change-Driven Incremental Build Algorithm

[3] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. “Scalable incremental building with dynamic task dependencies”. ASE 2018. DOI: 10.1145/3238147.3238196

PIE Observability

[4] Roelof Sol. Task "Observability in change driven incremental build systems with dynamic dependencies”. MSc thesis. Link: https://repository.tudelft.nl/islandora/object/uuid%3A3bd052ee-b8a0-4687-85d0-ca6df0b07d0d

PIE Implementation

[5] https://github.com/metaborg/pie/

PIE Future Work

Concurrency

(Automatic?) task parallelism

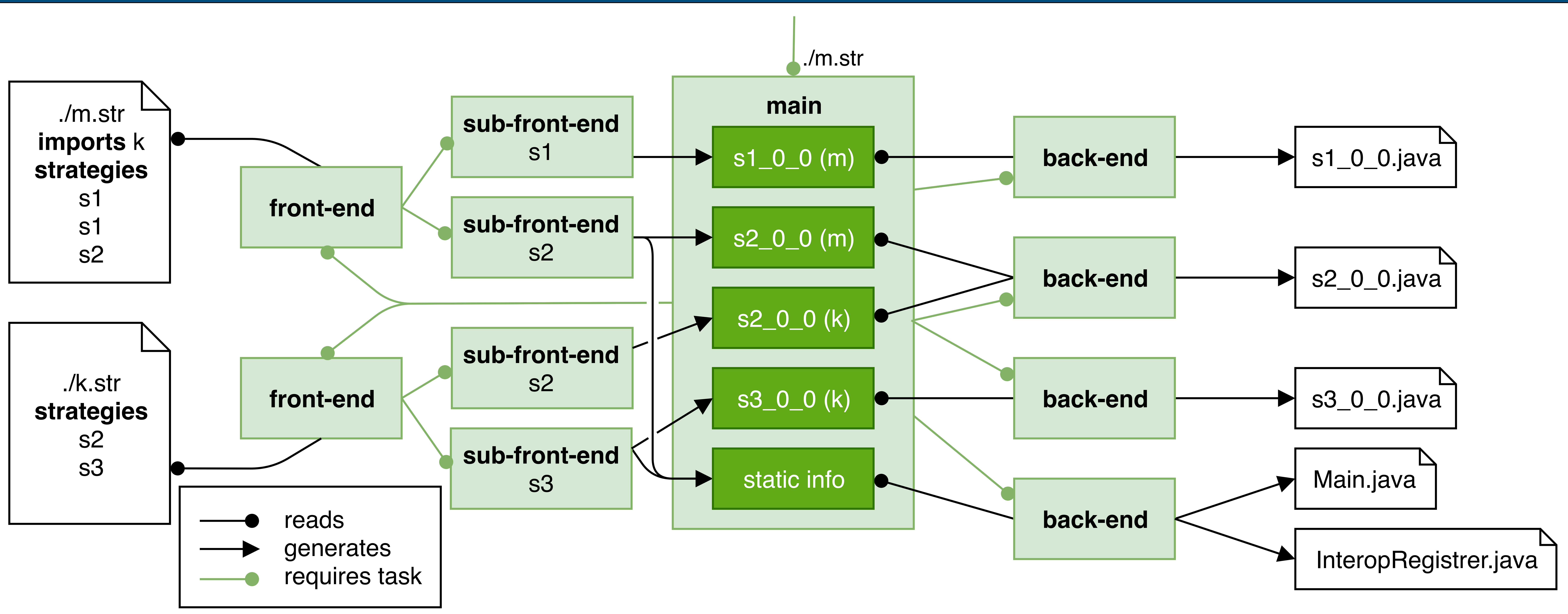
Asynchronous tasks

Distributed cache & distributed builds

Automatic deployment through partial application

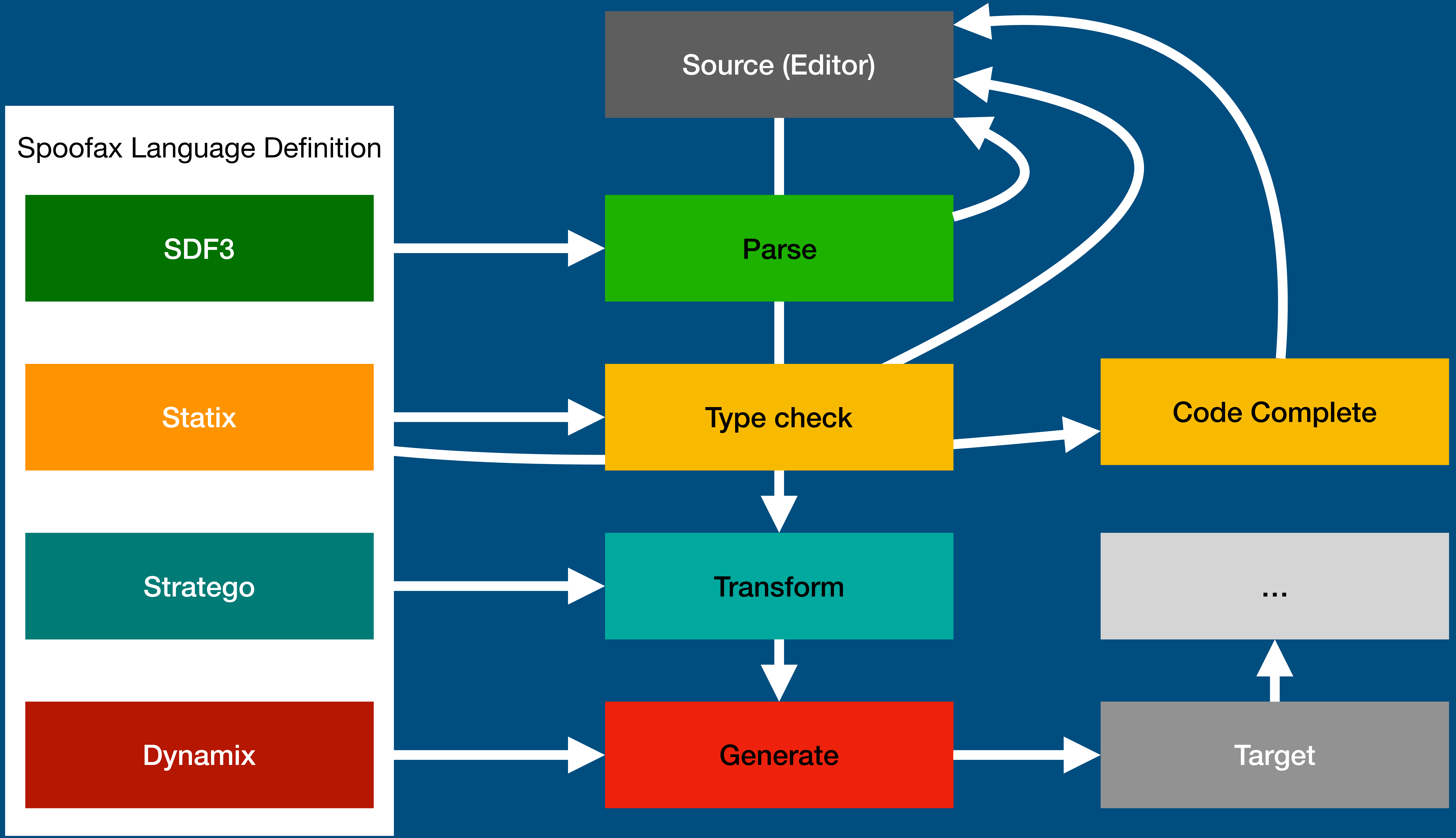
Cycle?

Ongoing Work: Spoofax3


./m.str
./m.str
imports k
strategies
s1
s1
s2
./k.str
strategies
s2
s3
front-end
front-end
sub-front-end
s1
sub-front-end
s2
sub-front-end
s2
sub-front-end
s3
main
s1_0_0 (m)
s2_0_0 (m)
s2_0_0 (k)
s3_0_0 (k)
static info
back-end
back-end
back-end
back-end
s1_0_0.java
s2_0_0.java
s3_0_0.java
Main.java
InteropRegistrer.java
reads
generates
requires task

Stratego

Transform



Source (Editor)
Spoofax Language Definition
SDF3
Statix
Stratego
Dynamix
Parse
Type check
Transform
Generate
Code Complete
...
Target

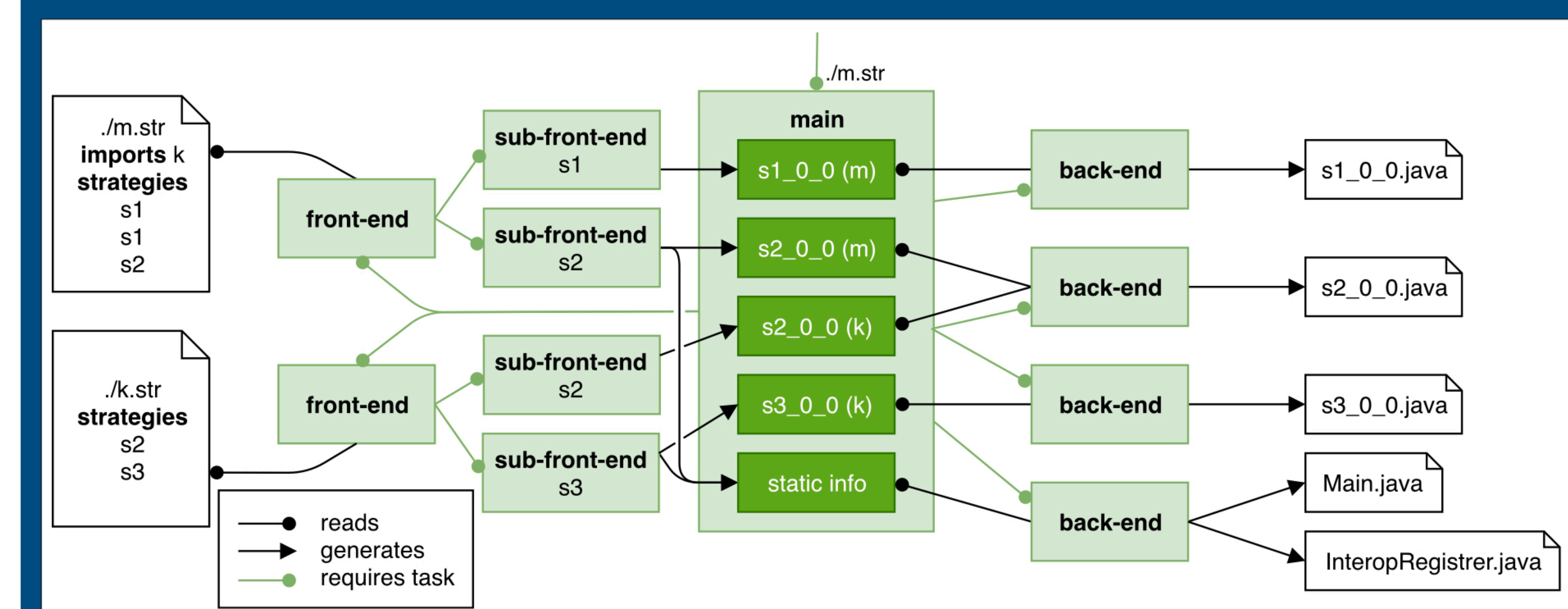
Conclusion

```
module lang/booleans/syntax

context-free syntax

  Type.BoolT = <Bool>

  Exp.Eq    = <<Exp> == <Exp>> {non-assoc}

  Exp.True  = <true>
  Exp.False = <false>
  Exp.Not   = <!<Exp>>
  Exp.And   = <<Exp> && <Exp>> {left}
  Exp.Or    = <<Exp> || <Exp>> {left}

  Exp.If = <
    if <Exp> then
      <Exp>
    else
      <Exp>
  >

  Exp.IfT = <
    if <Exp> then
      <Exp>
  >

  Val.BoolV = <<BoolV>>
  BoolV.True = <true>
  BoolV.False = <false>

context-free priorities

  Exp.Not > Exp.Eq > Exp.And
  > Exp.Or > Exp.IfT > Exp.If
```

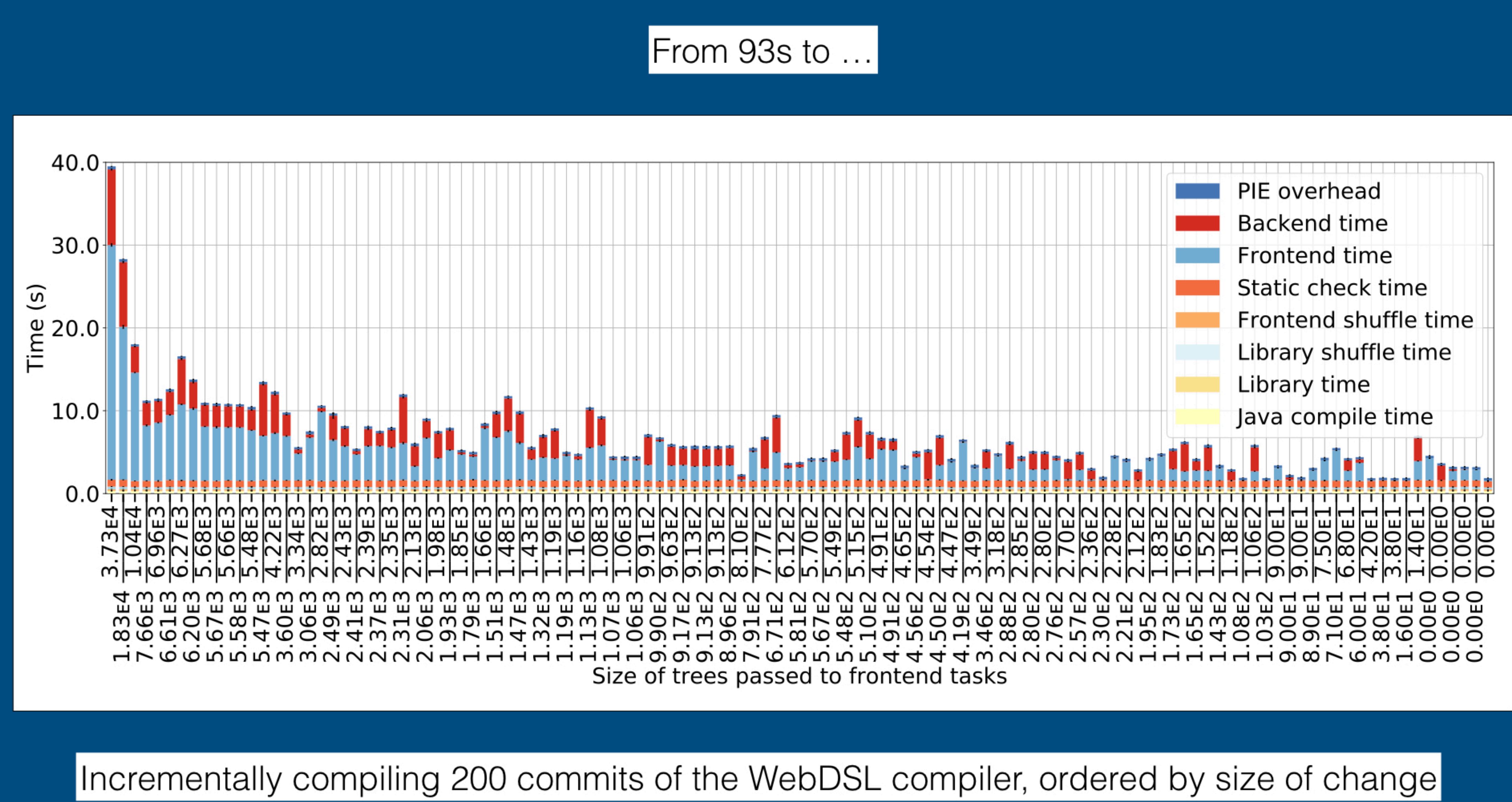
```
module lang/booleans/dynamics

imports signatures/lang/base/syntax-sig
imports signatures/lang/booleans/syntax-sig

rules

  eval(|f) :
    True() → BoolV(True())

  eval(|f) :
    False() → BoolV(False())

  eval(|f) :
    Not(e) → <notb>v
    with <eval(|f)> e ⇒ v

  eval(|f) :
    And(e1, e2) → <andb(|f)>(v1, e2)
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2

  eval(|f) :
    Or(e1, e2) → <orb(|f)>(v1, v2)
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2

  eval(|f) :
    If(e1, e2, e3) → v
    with <eval(|f)> e1 ⇒ v1
    with <ifb(|f)> (v1, e2, e3) ⇒ v

  eval(|f) :
    Eq(e1, e2) → v
    with <eval(|f)> e1 ⇒ v1
    with <eval(|f)> e2 ⇒ v2
    with <eqb> (e1, e2) ⇒ v
```

```
module lang/arithmetic/syntax

imports lang/base/syntax

context-free syntax // arithmetic

  Type.IntT = <Int>

  Exp.Int   = <<INT>>
  Exp.Add   = <<Exp> + <Exp>> {left}
  Exp.Sub   = <<Exp> - <Exp>> {left}
  Exp.Mul   = <<Exp> * <Exp>> {left}

  Val.IntV  = <<INT>>

context-free priorities

  Exp.Mul > {left: Exp.Add Exp.Sub}
```

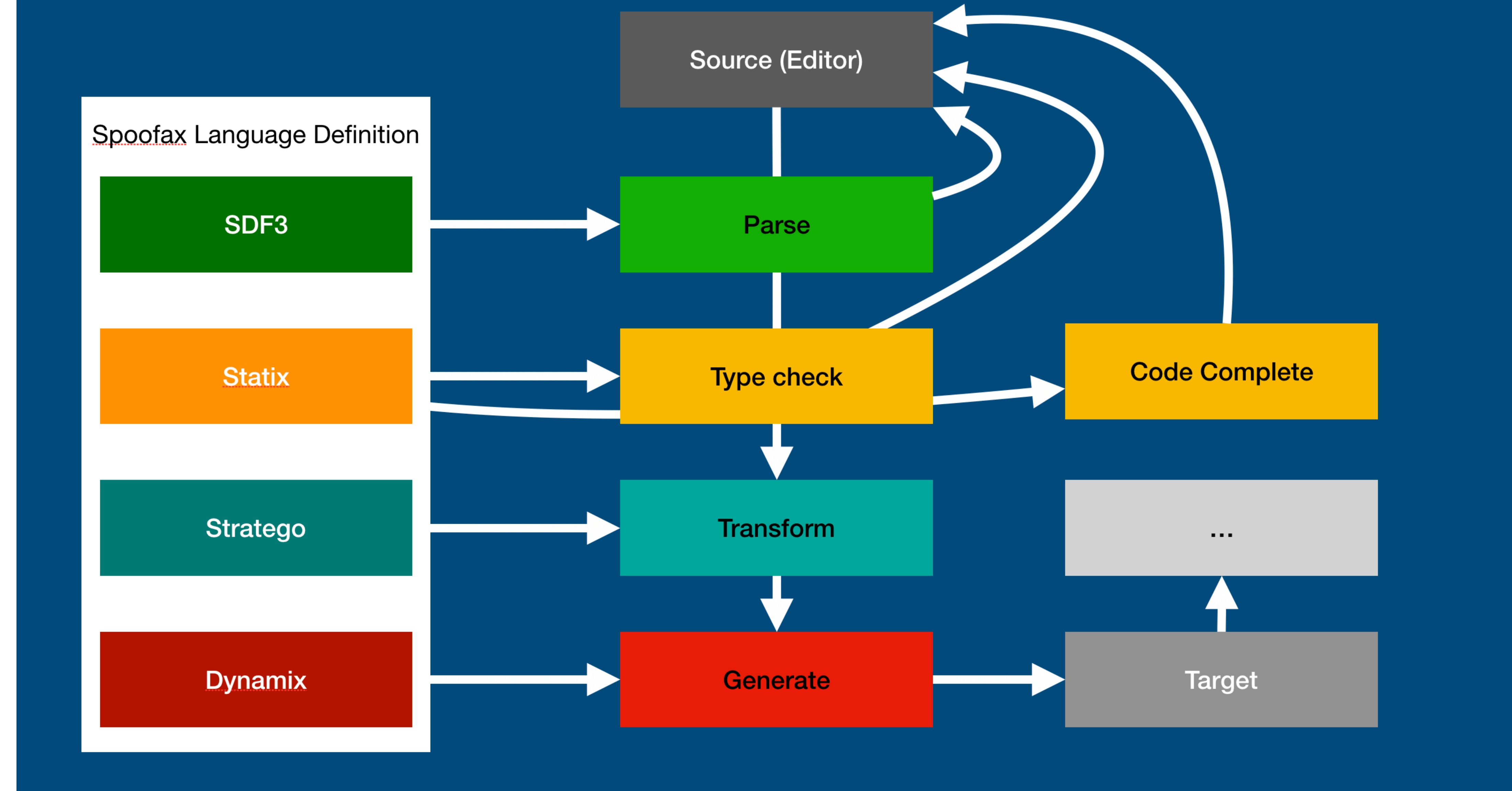
```
module lang/arithmetic/dynamics

imports signatures/lang/base/syntax-sig
imports signatures/lang/arithmetic/syntax-sig

imports lang/booleans/dynamics

signature
  constructors
    IntV : Int → Val

rules

  eval(|f) :
    Int(i) → IntV(i)

  eval(|f) :
    Add(e1, e2) → IntV(<addS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)

  eval(|f) :
    Sub(e1, e2) → IntV(<subtS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)

  eval(|f) :
    Mul(e1, e2) → IntV(<mulS>(i, j))
    with <eval(|f)> e1 ⇒ IntV(i)
    with <eval(|f)> e2 ⇒ IntV(j)
```



Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System
Jeff Smits, Gabriël Konat, Eelco Visser. To appear in <Programming> 2020



Incrementally compiling 200 commits of the WebDSL compiler, ordered by size of change



Build Automation and Programming Languages

BAPL 2020

Software building is an unloved but unavoidable part of the software engineering process, which requires reliable and incremental automation to deliver reproducible results rapidly and continuously. Build systems and programming languages have historically been mostly evolving independently of each other; indeed, build systems are often extra-linguistic (a prototypical example being Make), which makes them generally applicable but also unaware of the accurate dependencies induced by programs in a particular language. Language-specific build systems can use the knowledge of syntax and semantics to guarantee reliable builds and are gaining popularity but typically provide only rudimentary support for polyglot programming.

The goal of this workshop is to bring together build automation experts and language designers and implementers to explore the interaction of build automation and programming languages in systems for incremental analysis, building, testing, packaging, and deployment of software. It is time for build automation and programming languages to start evolving together, because language design affects the “buildability” of programs in a significant way and, conversely, build automation can benefits from the (static) semantics of languages to deliver faster and more reliable builds.

The scope of the workshop includes:

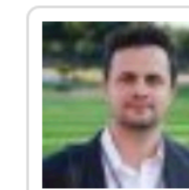
- Interaction between programming language design and build system design
- Build systems, both general-purpose and language-specific
- IDEs, particularly incremental program analysis
- Feedback-directed optimisation, where program building and analysis are interlinked
- Incremental computation DSLs, aimed at incrementalising general computation
- Computational complexity of build systems
- Software package-management systems

Important Dates

AoE (UTC-12h)

Sun 15 Mar 2020
Submission Deadline

Organizing Committee

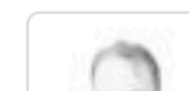


Andrey Mokhov
Jane Street
United Kingdom

Eelco Visser
Delft University of Technology
Netherlands

Program Committee

Ulf Adams
Google
Germany



Eelco Dolstra