Lecture 14: Interpreters

Eelco Visser

CS4200 Compiler Construction
TU Delft
December 2018

TU Delft

Reading Material

Design and implementation of DynSem, a domain-specific language for the definition of the dynamic semantics of programming languages.

DynSem provides facilities for implicitly passing memory (environment, store).

DynSem specifications can be compiled to or interpreted as interpreters for the languages they define.

RTA 2015

http://dx.doi.org/10.4230/LIPIcs.RTA.2015.365

DynSem: A DSL for Dynamic Semantics Specification

Vlad Vergu, Pierre Neron, and Eelco Visser

Delft University of Technology
Delft, The Netherlands
{v.a.vergu|p.j.m.neron|visser}@tudelft.nl

Abstract

The formal semantics of a programming language and its implementation are typically separately defined, with the risk of divergence such that properties of the formal semantics are not properties of the implementation. In this paper, we present DynSem, a domain-specific language for the specification of the dynamic semantics of programming languages that aims at supporting both formal reasoning and efficient interpretation. DynSem supports the specification of the *operational semantics* of a language by means of *statically typed conditional term reduction rules*. DynSem supports *concise* specification of reduction rules by providing *implicit build and match coercions* based on reduction arrows and implicit term constructors. DynSem supports *modular* specification by adopting implicit propagation of semantic components from I-MSOS, which allows omitting propagation of components such as environments and stores from rules that do not affect those. DynSem supports the declaration of *native operators* for delegation of aspects of the semantics to an external definition or implementation. DynSem supports the definition of auxiliary *meta-functions*, which can be expressed using regular reduction rules and are subject to semantic component propagation. DynSem specifications are *executable* through automatic generation of a Java-based AST interpreter.

1998 ACM Subject Classification F.3.2. Semantics of Programming Languages (D.3.1.)

Keywords and phrases programming languages, dynamic semantics, reduction semantics, semantics engineering, IDE, interpreters, modularity

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.365

1 Introduction

The specification of the dynamic semantics is the core of a programming language design as it describes the runtime behavior of programs. In practice, the implementation of a compiler or an interpreter for the language often stands as the *only* definition of the semantics of a language. Such implementations, in a traditional programming language, often lack the clarity and the conciseness that a specification in a formal semantics framework provides. Therefore, they are a poor source of *documentation* about the semantics. On the other hand, formal definitions are not executable to the point that they can be used as implementations to run programs. Even when both a formal specification and an implementation co-exist, they typically diverge. As a result, important properties of a language as established based on its formal semantics may not hold for its implementation. Our goal is to unify the semantics engineering and language engineering of programming language designs [22] by providing a notation for the specification of the dynamic semantics that can serve both as a readable formalization as well as the source of an execution engine.

In this paper, we present DynSem, a DSL for the concise, modular, statically typed, and executable specification of the dynamic semantics of programming languages. DynSem

26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 365–378
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This paper describes the partial evaluation of the DynSem meta-interpreter with an object program.

The implementation makes use of Truffle a framework for run-time (online) partial evaluation.

Truffle gets most of it benefit from Graal, an implementation of the JVM with special support for partial evaluation.

ManLang 2018

https://doi.org/10.1145/3237009.3237018

Specializing a Meta-Interpreter

JIT Compilation of DynSem Specifications on the Graal VM

Vlad Vergu
TU Delft
The Netherlands
v.a.vergu@tudelft.nl

Eelco Visser
TU Delft
The Netherlands
visser@acm.org

ABSTRACT

DynSem is a domain-specific language for concise specification of the dynamic semantics of programming languages, aimed at rapid experimentation and evolution of language designs. DynSem specifications can be executed to interpret programs in the language under development. To enable fast turnaround during language development, we have developed a meta-interpreter for DynSem specifications, which requires minimal processing of the specification. In addition to fast development time, we also aim to achieve fast run times for interpreted programs.

In this paper we present the design of a meta-interpreter for DynSem and report on experiments with JIT compiling the application of the meta-interpreter on the Graal VM. By interpreting specifications directly, we have minimal compilation overhead. By specializing pattern matches, maintaining call-site dispatch chains and using native control-flow constructs we gain significant run-time performance. We evaluate the performance of the meta-interpreter when applied to the Tiger language specification running a set of common benchmark programs. Specialization enables the Graal VM to JIT compile the meta-interpreter giving speedups of up to factor 15 over running on the standard Oracle Java VM.

CCS CONCEPTS

• **Software and its engineering** → **Interpreters**; **Domain specific languages**; *Semantics*;

KEYWORDS

dynamic semantics, interpretation, JIT, run-time optimization

ACM Reference Format:
Vlad Vergu and Eelco Visser. 2018. Specializing a Meta-Interpreter: JIT Compilation of DynSem Specifications on the Graal VM. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3237009.3237018

1 INTRODUCTION

The dynamic semantics of a programming language defines the run time execution behavior of programs in the language. Ideally, the design of a programming language starts with the specification of its dynamic semantics to provide a high-level readable and unambiguous definition. However, understanding the design of a programming language also requires experimentation by actually running programs. Therefore, this ideal route is rarely taken, but language designs are embodied in the implementation of interpreters or compilers instead.

We have previously designed DynSem [33], a high-level meta-DSL for dynamic semantics specifications of programming languages, with the aim of supporting readable *and* executable specification. It supports the definition of modular and concise semantics by means of reduction rules with implicit propagation of contextual information. DynSem's executable semantics entails that specifications can be used to interpret object language programs.

In our early prototypes, DynSem specifications were compiled to an interpreter. The process of generating a Java implementation of an interpreter and compiling that generated code caused long turnaround times during language prototyping. In order to support rapid prototyping with short turnaround times, we turned to interpreting specifications directly instead of compiling them. A DynSem interpreter is a *meta-interpreter* since the programs it interprets are themselves interpreters. Figure 1 depicts the high-level architecture of the DynSem meta-interpreter. First, a DynSem specification is desugared (explicated) to make implicit passing of semantic components explicit. The resulting specification in DynSem Core is then loaded into the meta-interpreter together with the AST of the interpreted object program. The interpreter consumes the program as input enacting the specification. This produces the desired result of a short turnaround time for experimenting with dynamic semantics specifications.

Meta-interpretation reduces the turnaround time at the expense of execution performance. At run time there are two interpreter layers operating (the meta-language interpreter and the object-language interpreter) which introduces substantial overhead. While we envision DynSem as a convenient way to prototype the dynamic semantics of programming languages, ultimately we also envision it as a convenient way to bridge the gap between the prototyping and production phases of a programming language's lifecycle. Thus, we not only want an interpreter fast, but we also want a fast interpreter, which raises the question: Can we achieve fast object-language interpreters by optimizing the meta-interpretation of dynamic semantics specifications?

Direct vanilla interpreters are in general slow to begin with, even when they are implemented in a host language that is JIT-ed. This is because the host JIT is unable to see patterns in the object language and to meaningfully optimize the interpreter. The task of optimizing an interpreter has traditionally been long and

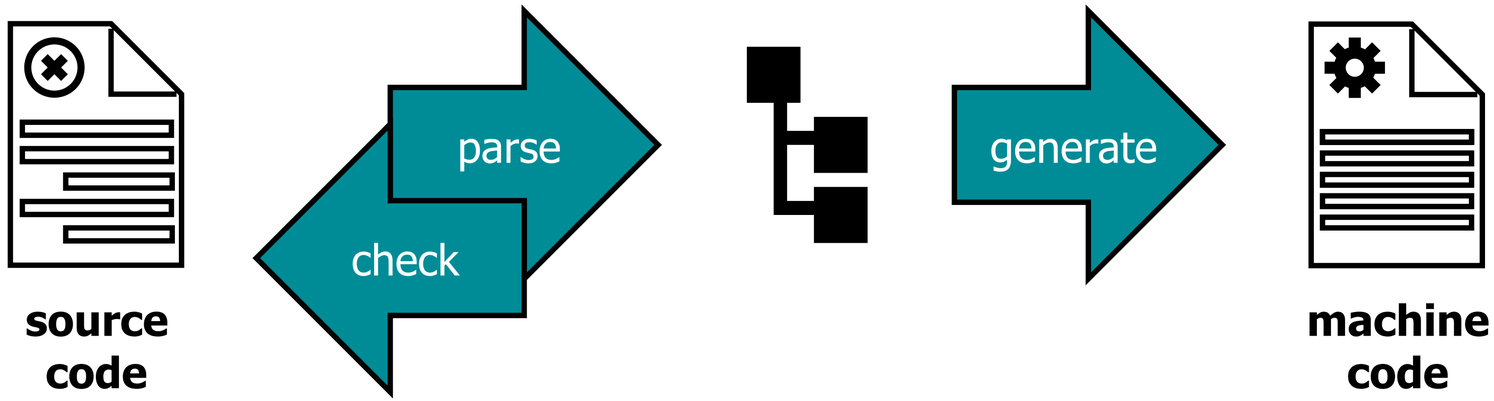
ManLang'18, September 12–14, 2018, Linz, Austria

ACM ISBN 978-1-4503-6424-9/18/09…$15.00
https://doi.org/10.1145/3237009.3237018

Semantics

What is the meaning of a program?

`meaning(p) = behavior(p)`



`meaning(p)` = what happens when executing the generated (byte) code to which p is compiled

What is the meaning of a program?

```
meaning(p) = behavior(p)
```

What *is* behavior?

How can we *observe* behavior?

Mapping input to output

Changes to state of the system

Which behavior is essential, which accidental?

How can we define the semantics of a program?

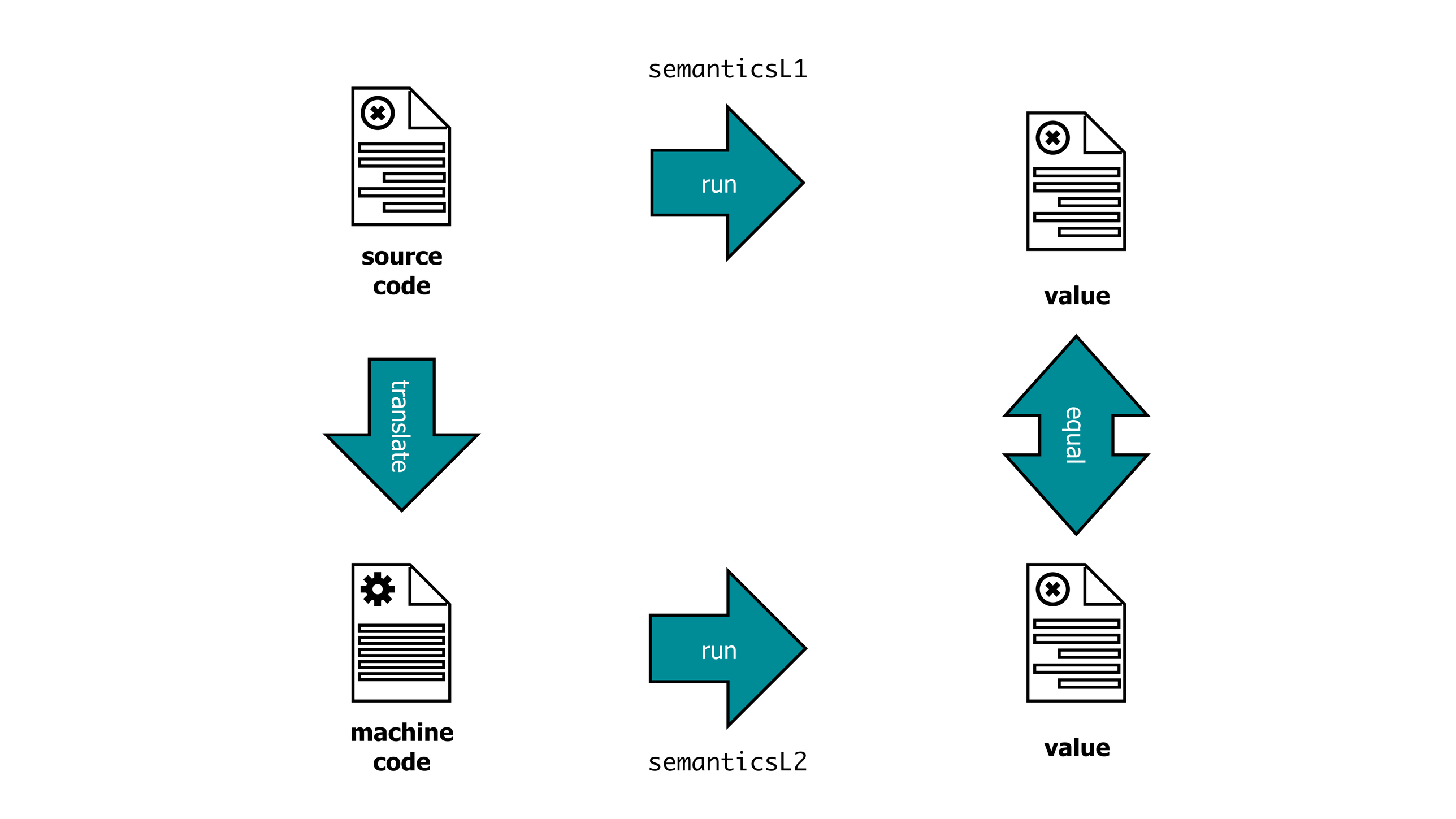
Compiler defines *translational* semantics

```
semanticsL1(p) = semanticsL2(translate(p))
```

Requires understanding `translate` and `semanticsL2`

How do we know that `translate` is correct?

Is there a more ***direct description*** of `semanticsL1`?


semanticsL1
run
source code
value
translate
equal
run
machine code
semanticsL2
value

Verifying Compiler Correctness

Direct semantics of source language provides a specification

How to check correctness?

Testing: for ***many*** programs p (and inputs i) ***test*** that

```
run(p)(i)  == run(translate(p))(i)
```

Verification: for ***all*** programs p (and inputs i) ***prove*** that

```
run(p)(i)  == run(translate(p))(i)
```

Validating Semantics

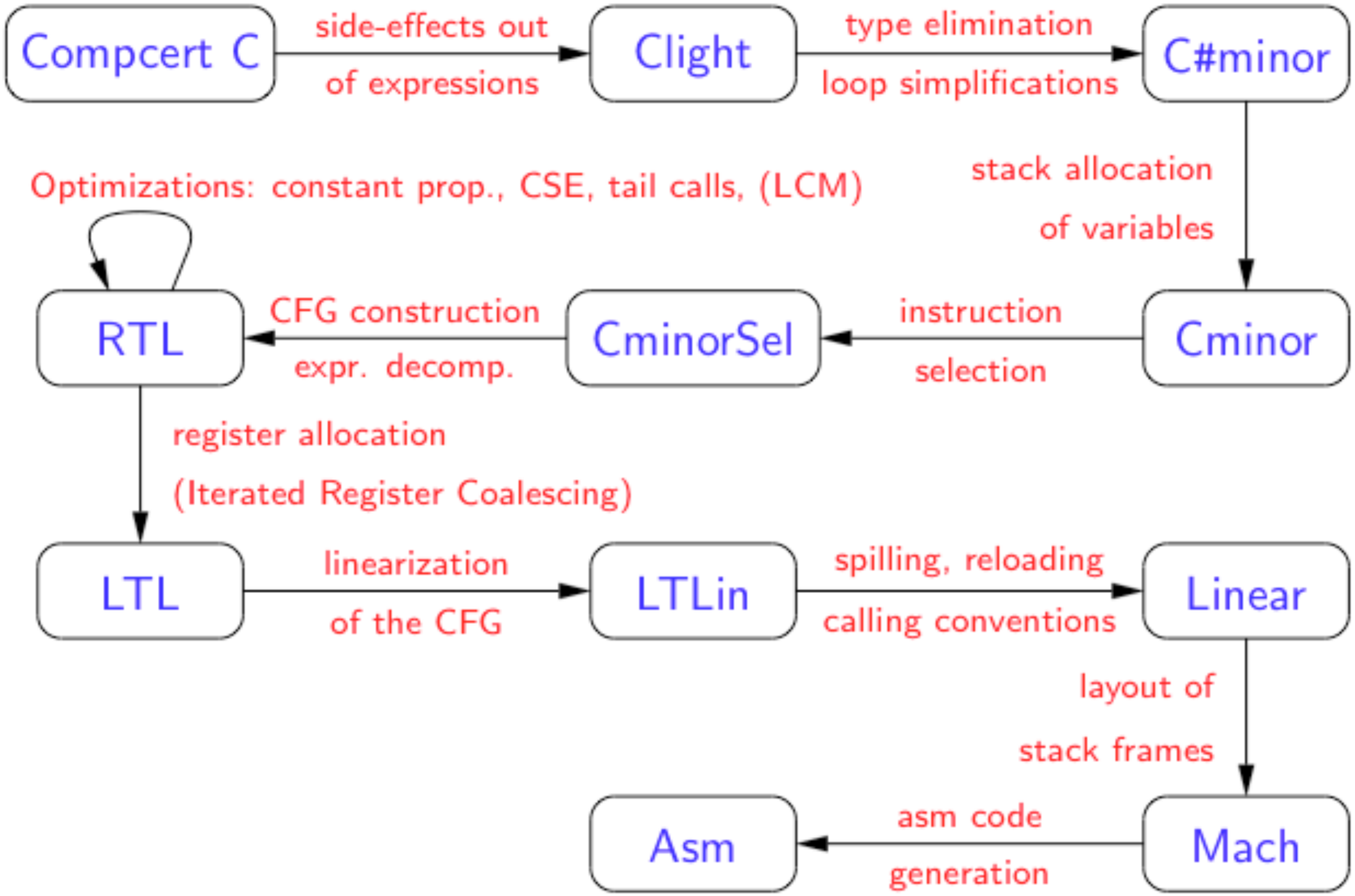
Is this the right semantics?

Testing: for ***many*** programs p (and inputs i) ***test*** that

```
run(p)(i)  ==  v
```

Requires specifying desired <p, i, v> combinations
(aka unit testing)

The CompCert C Compiler



http://compcert.inria.fr/compcert-C.html

Compiler Construction Courses of the Future

Language Specification

syntax definition
name binding
type system
dynamic semantics
translation
transformation

safety properties

Language Implementation

generating implementations
from specifications

parser generation
constraint resolution
partial evaluation
…

Language Verification

proving correctness

Language Testing

test generation

Operational Semantics

Operational Semantics

What is the result of execution of a program?

- How is that result achieved?

Natural Semantics

- How is overall result of execution obtained?

Structural Operational Semantics

- What are the individual steps of an execution?

Defined using a transition system

DynSem

Design and implementation of DynSem, a domain-specific language for the definition of the dynamic semantics of programming languages.

DynSem provides facilities for implicitly passing memory (environment, store).

DynSem specifications can be compiled to or interpreted as interpreters for the languages they define.

RTA 2015

http://dx.doi.org/10.4230/LIPIcs.RTA.2015.365

DynSem: A DSL for Dynamic Semantics Specification

Vlad Vergu, Pierre Neron, and Eelco Visser

Delft University of Technology
Delft, The Netherlands
{v.a.vergu|p.j.m.neron|visser}@tudelft.nl

Abstract

The formal semantics of a programming language and its implementation are typically separately defined, with the risk of divergence such that properties of the formal semantics are not properties of the implementation. In this paper, we present DynSem, a domain-specific language for the specification of the dynamic semantics of programming languages that aims at supporting both formal reasoning and efficient interpretation. DynSem supports the specification of the *operational semantics* of a language by means of *statically typed conditional term reduction rules*. DynSem supports *concise* specification of reduction rules by providing *implicit build and match coercions* based on reduction arrows and implicit term constructors. DynSem supports *modular* specification by adopting implicit propagation of semantic components from I-MSOS, which allows omitting propagation of components such as environments and stores from rules that do not affect those. DynSem supports the declaration of *native operators* for delegation of aspects of the semantics to an external definition or implementation. DynSem supports the definition of auxiliary *meta-functions*, which can be expressed using regular reduction rules and are subject to semantic component propagation. DynSem specifications are *executable* through automatic generation of a Java-based AST interpreter.

1998 ACM Subject Classification F.3.2. Semantics of Programming Languages (D.3.1.)

Keywords and phrases programming languages, dynamic semantics, reduction semantics, semantics engineering, IDE, interpreters, modularity

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.365

1 Introduction

The specification of the dynamic semantics is the core of a programming language design as it describes the runtime behavior of programs. In practice, the implementation of a compiler or an interpreter for the language often stands as the *only* definition of the semantics of a language. Such implementations, in a traditional programming language, often lack the clarity and the conciseness that a specification in a formal semantics framework provides. Therefore, they are a poor source of *documentation* about the semantics. On the other hand, formal definitions are not executable to the point that they can be used as implementations to run programs. Even when both a formal specification and an implementation co-exist, they typically diverge. As a result, important properties of a language as established based on its formal semantics may not hold for its implementation. Our goal is to unify the semantics engineering and language engineering of programming language designs [22] by providing a notation for the specification of the dynamic semantics that can serve both as a readable formalization as well as the source of an execution engine.

In this paper, we present DynSem, a DSL for the concise, modular, statically typed, and executable specification of the dynamic semantics of programming languages. DynSem

26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 365–378
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Interpreters for Spoofax Languages

gcdAB.pbox

```
let
  gcd = box(0)
in
  let f =
      fun (a, b) {
        if (b == 0)
          a
        else
          unbox(gcd)(b, a % b)
        end
      }
  in
    setbox(gcd, f);
    unbox(gcd)(1134903170, 1836)
  end
end
```

gcdAB.aterm

```
Let(
  [Bind("gcd", Box(Num("0")))]
, Let(
    [ Bind(
        "f"
      , Fun(
          ["a", "b"]
        , If(
            Eq(Var("b"), Num("0"))
          , Var("a")
          , App(
              Unbox(Var("gcd"))
            , [Var("b"), Mod(Var("a"), Var("b"))]
            )
          )
        )
      )
    ]
  , Seq(
      SetBox(Var("gcd"), Var("f"))
    , App(
        Unbox(Var("gcd"))
      , [Num("1134903170"), Num("1836")]
      )
    )
  )
)
```

gcdAB.evaluated.aterm

```
R_default_V(
  NumV(34)
, Map(
    "Store"
  , Bind(923, RefV(925))
  , Bind(
      925
    , ObjV(
        Map(
          "Env"
        , Bind("outer", 922)
        , Bind("self", 923)
        , Bind("super", 924)
        )
      )
    )
  , Bind(
      926
    , ClosV(
        Fun(
          ["a", "b"]
        , If(
            Eq(Var("b"), Num("0"))
          , Var("a")
          , App(
              Unbox(Var("gcd"))
            , [Var("b"), Mod(Var("a"), V
            )
```

Example: DynSem Semantics of PAPL-Box

```
let
  fac = box(0)
in
  let f = fun (n) {
        if (n == 0)
          1
        else
          n * (unbox(fac) (n - 1))
        end
     }
  in
    setbox(fac, f);
    unbox(fac)(10)
   end
end
```

Features
- Arithmetic
- Booleans
- Comparisons
- Mutable variables
- Functions
- Boxes

Components
- Syntax in SDF3
- Dynamic Semantics in DynSem

Abstract Syntax from Concrete Syntax

```
module Arithmetic

imports Expressions
imports Common

context-free syntax

  Expr.Num   = INT

  Expr.Plus  = [[Expr] + [Expr]] {left}
  Expr.Minus = [[Expr] - [Expr]] {left}
  Expr.Times = [[Expr] * [Expr]] {left}
  Expr.Mod   = [[Expr] % [Expr]] {left}

context-free priorities

    {left: Expr.Times Expr.Mod }
  > {left: Expr.Minus Expr.Plus }
```

```
module Arithmetic-sig

imports Expressions-sig
imports Common-sig

signature
    sorts Expr
    constructors
      Num   : INT -> Expr
      Plus  : Expr * Expr -> Expr
      Minus : Expr * Expr -> Expr
      Times : Expr * Expr -> Expr
      Mod   : Expr * Expr -> Expr
```

src-gen/ds-signatures/Arithmetic-sig

Values, Meta-Variables, and Arrows

```
module values

signature
  sorts V Unit
  constructors
    U : Unit
  variables
    v: V
```

```
module expressions

imports values
imports Expressions-sig

signature
  arrows
    Expr --> V
  variables
    e : Expr
    x : String
```

Term Reduction Rules

```
module arithmetic-explicit

imports expressions primitives Arithmetic-sig

signature
  constructors
    NumV: Int -> V

rules

  Num(__String2INT__(n)) --> NumV(str2int(n)).

  Plus(e1, e2) --> NumV(plusI(i1, i2))
  where
    e1 --> NumV(i1); e2 --> NumV(i2).

  Minus(e1, e2) --> NumV(minusI(i1, i2))
  where
    e1 --> NumV(i1); e2 --> NumV(i2).
```

```
module expressions

imports values
imports Expressions-sig

signature
  arrows
    Expr --> V
  variables
    e : Expr
    x : String
```

Native Operations

```
module arithmetic-explicit

imports expressions primitives Arithmetic-sig

signature
  constructors
    NumV: Int -> V

rules

  Num(__String2INT__(n)) --> NumV(str2int(n)).

  Plus(e1, e2) --> NumV(plusI(i1, i2))
  where
    e1 --> NumV(i1); e2 --> NumV(i2).

  Minus(e1, e2) --> NumV(minusI(i1, i2))
  where
    e1 --> NumV(i1); e2 --> NumV(i2).
```

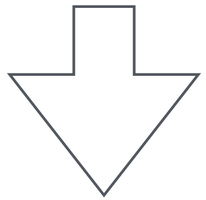
```
module primitives
signature
  native operators
    str2int : String -> Int
    plusI   : Int * Int -> Int
    minusI  : Int * Int -> Int
```

```
public class Natives {

   public static int plusI_2(int i1, int i2) {
      return i1 + i2;
   }

   public static int str2int_1(String s) {
      return Integer.parseInt(s);
   }
}
```

Arrows as Coercions

```
rules

  Plus(NumV(i1), NumV(i2)) --> NumV(plusI(i1, i2)).
```

```
signature
  constructors
    Plus : Expr * Expr -> Expr
    NumV : Int -> V
  arrows
    Expr --> V
```



```
rules

  Plus(e1, e2) --> NumV(plusI(i1, i2))
  where
    e1 --> NumV(i1);
    e2 --> NumV(i2).
```

Modular

```
module arithmetic

imports Arithmetic-sig
imports expressions
imports primitives

signature
  constructors
    NumV: Int -> V

rules

  Num(str) --> NumV(str2int(str)).

  Plus(NumV(i1), NumV(i2)) --> NumV(plusI(i1, i2)).

  Minus(NumV(i1), NumV(i2)) --> NumV(minusI(i1, i2)).

  Times(NumV(i1), NumV(i2)) --> NumV(timesI(i1, i2)).

  Mod(NumV(i1), NumV(i2)) --> NumV(modI(i1, i2)).
```

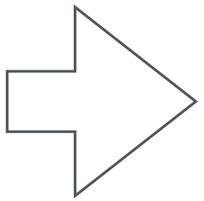
```
module boolean

imports Booleans-sig expressions

signature
  constructors
    BoolV : Bool -> V

rules

  True() --> BoolV(true).
  False() --> BoolV(false).

  Not(BoolV(false)) --> BoolV(true).
  Not(BoolV(true)) --> BoolV(false).

  Or(BoolV(true), _) --> BoolV(true).
  Or(BoolV(false), e) --> e.

  And(BoolV(false), _) --> BoolV(false).
  And(BoolV(true), e) --> e.
```

```
module comparison

imports Comparisons-sig arithmetic boolean

rules

  Gt(NumV(i1), NumV(i2)) --> BoolV(gtI(i1, i2)).

  Eq(NumV(i1), NumV(i2)) --> BoolV(eqI(i1, i2)).

  Eq(BoolV(b1), BoolV(b2)) --> BoolV(eqB(b1, b2)).
```

Interpreter is not defined as a single match over sort

Control-Flow

```
module controlflow

imports ControlFlow-sig
imports expressions
imports boolean

rules

  Seq(v, e2) --> e2.

  If(BoolV(true), e1, _) --> e1.

  If(BoolV(false), _, e2) --> e2.
```



```
module controlflow

imports ControlFlow-sig
imports expressions
imports boolean

rules

  Seq(e1, e2) --> v2
  where
    e1 --> v1;
    e2 --> v2.

  If(e1, e2, e3) --> v
  where
    e1 --> BoolV(true);
    e2 --> v.

  If(e1, e2, e3) --> v
  where
    e1 --> BoolV(false);
    e3 --> v.
```

Immutable Variables: Environment Passing

```
constructors
  Let : ID * Expr * Expr -> Expr
  Var : ID -> Expr
```

```
module variables

imports Variables-sig environment

rules

  E |- Let(x, v: V, e2) --> v2
  where
    Env {x |--> v, E} |- e2 --> v2.

  E |- Var(x) --> E[x].
```

```
module environment

imports values

signature
  sort aliases
    Env = Map<String, V>
  variables
    E : Env
```

First-Class Functions: Environment in Closure

```
constructors
  Fun : ID * Expr -> Expr
  App : Expr * Expr -> Expr
```

```
module unary-functions

imports expressions environment

signature
  constructors
    ClosV : String * Expr * Env -> V
rules

  E |- Fun(x, e) --> ClosV(x, e, E).

  E |- App(e1, e2) --> v
  where
    E |- e1 --> ClosV(x, e, E');
    E |- e2 --> v2;
    Env {x |--> v2, E'} |- e --> v.
```

```
module environment

imports values

signature
  sort aliases
    Env = Map<String, V>
  variables
    E : Env
```

Implicit Propagation

```
rules

  Plus(NumV(i1), NumV(i2)) --> NumV(plusI(i1, i2)).
```

```
rules

  Plus(e1, e2) --> NumV(plusI(i1, i2))
  where
    e1 --> NumV(i1);
    e2 --> NumV(i2).
```

```
rules

  E |- Plus(e1, e2) --> NumV(plusI(i1, i2))
  where
    E |- e1 --> NumV(i1);
    E |- e2 --> NumV(i2).
```

Mutable Boxes: Store

```
module box
imports store arithmetic
signature
  constructors
    Box    : Expr -> Expr
    Unbox  : Expr -> Expr
    SetBox : Expr * Expr -> Expr
  constructors
    BoxV: Int -> V
rules

  Box(e) :: S --> BoxV(loc) :: Store {loc |--> v, S'}
  where e :: S --> v :: S';
        fresh => loc.

  Unbox(BoxV(loc)) :: S --> S[loc].

  SetBox(BoxV(loc), v) :: S --> v :: Store {loc |--> v, S}.
```

```
module store

imports values

signature
  sort aliases
   Store = Map<Int, V>
  variables
    S : Store
```

Mutable Variables: Environment + Store

```
constructors
  Let : ID * Expr * Expr -> Expr
  Var : ID -> Expr
  Set : String * Expr -> Expr
```

```
module variables-mutable
imports Variables-sig store
rules

  E |- Var(x) :: S --> v :: S
  where E[x] => loc; S[loc] => v.

  E |- Let(x, v, e2) :: S1 --> v2 :: S3
  where
    fresh => loc;
    {loc |--> v, S1} => S2;
    Env {x |--> loc, E} |- e2 :: S2 --> v2 :: S3.

  E |- Set(x, v) :: S --> v :: Store {loc |--> v, S}
  where E[x] => loc.
```

```
module store

imports values

signature
  sort aliases
    Env = Map<ID, Int>
    Store = Map<Int, V>
  variables
    E : Env
    S : Store
```

Implicit Store Threading

```
rules

  Plus(NumV(i1), NumV(i2)) --> NumV(plusI(i1, i2)).
```

```
rules

  Plus(e1, e2) --> NumV(plusI(i1, i2))
  where
    e1 --> NumV(i1);
    e2 --> NumV(i2).
```

```
rules

  E |- Plus(e1, e2) :: S1 --> NumV(plusI(i1, i2)) :: S3
  where
    E |- e1 :: S1 --> NumV(i1) :: S2;
    E |- e2 :: S2 --> NumV(i2) :: S3.
```

Abstraction: Env/Store Meta Functions

```
module store

imports values

signature
  sort aliases
    Env = Map<String, Int>
    Store = Map<Int, V>
  variables
    E : Env
    S : Store
  arrows
    readVar  : String --> V
    bindVar  : String * V --> Env
    writeVar : String * V --> V

    allocate : V --> Int
    write    : Int * V --> V
    read     : Int --> V
```

```
rules

  allocate(v) --> loc
  where
    fresh => loc;
    write(loc, v) --> _.

  write(loc, v) :: S -->
    v :: Store {loc |--> v, S}.

  read(loc) :: S --> S[loc] :: S.

rules

  bindVar(x, v) --> {x |--> loc}
  where allocate(v) --> loc.

  E |- readVar(x) --> read(E[x]).

  E |- writeVar(x, v) --> write(E[x], v).
```

Boxes with Env/Store Meta Functions

```
module boxes

signature
  constructors
    Box    : Expr -> Expr
    Unbox  : Expr -> Expr
    SetBox : Expr * Expr -> Expr
  constructors
    BoxV: V -> V

rules

  Box(v) --> BoxV(NumV(allocate(v))).

  Unbox(BoxV(NumV(loc))) --> read(loc).

  SetBox(BoxV(NumV(loc)), v) --> write(loc, v).
```

Mutable Variables with Env/Store Meta Functions

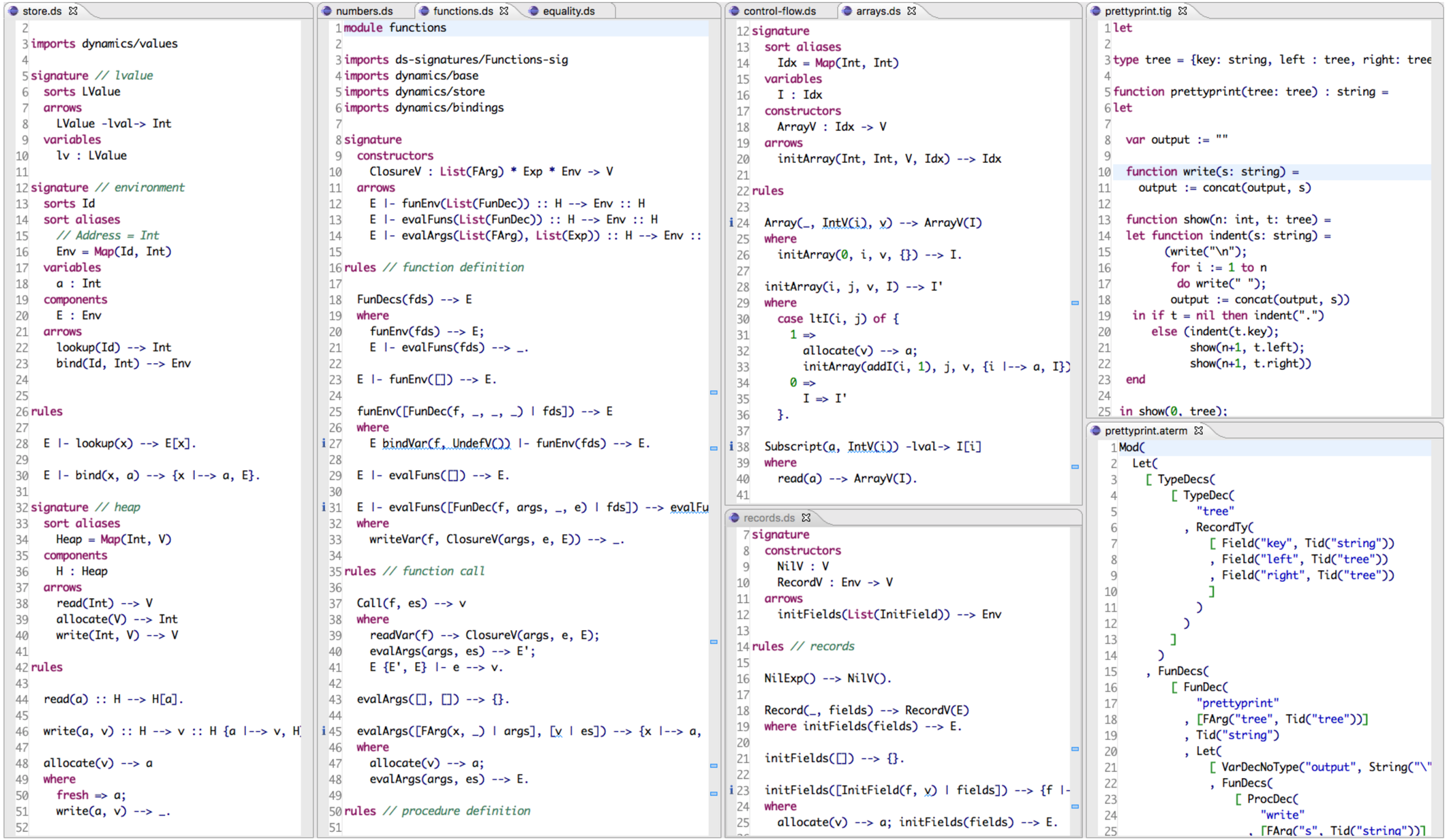
```
constructors
  Let : String * Expr * Expr -> Expr
  Var : String -> Expr
  Set : String * Expr -> Expr
```

```
module variables

imports expressions store

rules

  Var(x) --> readVar(x).

  E |- Let(x, v1, e) --> v2
  where
    bindVar(x, v1) --> E';
    Env {E', E} |- e --> v2.

  Set(x, v) --> v
  where
    writeVar(x, v) --> _.
```

Functions with Multiple Arguments

```
module functions

imports Functions-sig
imports variables

signature
  constructors
    ClosV    : List(ID) * Expr * Env -> V
    bindArgs : List(ID) * List(Expr) --> Env

rules

  E |- Fun(xs, e) --> ClosV(xs, e, E).

  App(ClosV(xs, e_body, E_clos), es) --> v'
  where
    bindArgs(xs, es) --> E_params;
    Env {E_params, E_clos} |- e_body --> v'.

  bindArgs([], []) --> {}.

  bindArgs([x | xs], [e | es]) --> {E, E'}
  where
    bindVar(x, e) --> E;
    bindArgs(xs, es) --> E'.
```

Tiger in DynSem



store.ds

```
imports dynamics/values

signature // lvalue
  sorts LValue
  arrows
    LValue -lval-> Int
  variables
    lv : LValue

signature // environment
  sorts Id
  sort aliases
    // Address = Int
    Env = Map(Id, Int)
  variables
    a : Int
  components
    E : Env
  arrows
    lookup(Id) --> Int
    bind(Id, Int) --> Env


rules

  E |- lookup(x) --> E[x].

  E |- bind(x, a) --> {x |--> a, E}.

signature // heap
  sort aliases
    Heap = Map(Int, V)
  components
    H : Heap
  arrows
    read(Int) --> V
    allocate(V) --> Int
    write(Int, V) --> V

rules

  read(a) :: H --> H[a].

  write(a, v) :: H --> v :: H {a |--> v, H

  allocate(v) --> a
  where
    fresh => a;
    write(a, v) --> _.
```

numbers.ds / functions.ds / equality.ds

```
module functions

imports ds-signatures/Functions-sig
imports dynamics/base
imports dynamics/store
imports dynamics/bindings

signature
  constructors
    ClosureV : List(FArg) * Exp * Env -> V
  arrows
    E |- funEnv(List(FunDec)) :: H --> Env :: H
    E |- evalFuns(List(FunDec)) :: H --> Env :: H
    E |- evalArgs(List(FArg), List(Exp)) :: H --> Env ::

rules // function definition

  FunDecs(fds) --> E
  where
    funEnv(fds) --> E;
    E |- evalFuns(fds) --> _.

  E |- funEnv([]) --> E.

  funEnv([FunDec(f, _, _, _) | fds]) --> E
  where
    E bindVar(f, UndefV()) |- funEnv(fds) --> E.

  E |- evalFuns([]) --> E.

  E |- evalFuns([FunDec(f, args, _, e) | fds]) --> evalFu
  where
    writeVar(f, ClosureV(args, e, E)) --> _.

rules // function call

  Call(f, es) --> v
  where
    readVar(f) --> ClosureV(args, e, E);
    evalArgs(args, es) --> E';
    E {E', E} |- e --> v.

  evalArgs([], []) --> {}.

  evalArgs([FArg(x, _) | args], [v | es]) --> {x |--> a,
  where
    allocate(v) --> a;
    evalArgs(args, es) --> E.

rules // procedure definition
```

control-flow.ds / arrays.ds

```
signature
  sort aliases
    Idx = Map(Int, Int)
  variables
    I : Idx
  constructors
    ArrayV : Idx -> V
  arrows
    initArray(Int, Int, V, Idx) --> Idx

rules

  Array(_, IntV(i), v) --> ArrayV(I)
  where
    initArray(0, i, v, {}) --> I.

  initArray(i, j, v, I) --> I'
  where
    case ltI(i, j) of {
      1 =>
        allocate(v) --> a;
        initArray(addI(i, 1), j, v, {i |--> a, I}
      0 =>
        I => I'
    }.

  Subscript(a, IntV(i)) -lval-> I[i]
  where
    read(a) --> ArrayV(I).
```

records.ds

```
signature
  constructors
    NilV : V
    RecordV : Env -> V
  arrows
    initFields(List(InitField)) --> Env

rules // records

  NilExp() --> NilV().

  Record(_, fields) --> RecordV(E)
  where initFields(fields) --> E.

  initFields([]) --> {}.

  initFields([InitField(f, v) | fields]) --> {f |-
  where
    allocate(v) --> a; initFields(fields) --> E.
```

prettyprint.tig

```
let

type tree = {key: string, left : tree, right: tree

function prettyprint(tree: tree) : string =
let

  var output := ""

  function write(s: string) =
    output := concat(output, s)

  function show(n: int, t: tree) =
  let function indent(s: string) =
        (write("\n");
         for i := 1 to n
          do write(" ");
         output := concat(output, s))
   in if t = nil then indent(".")
      else (indent(t.key);
            show(n+1, t.left);
            show(n+1, t.right))
  end

in show(0, tree);
```

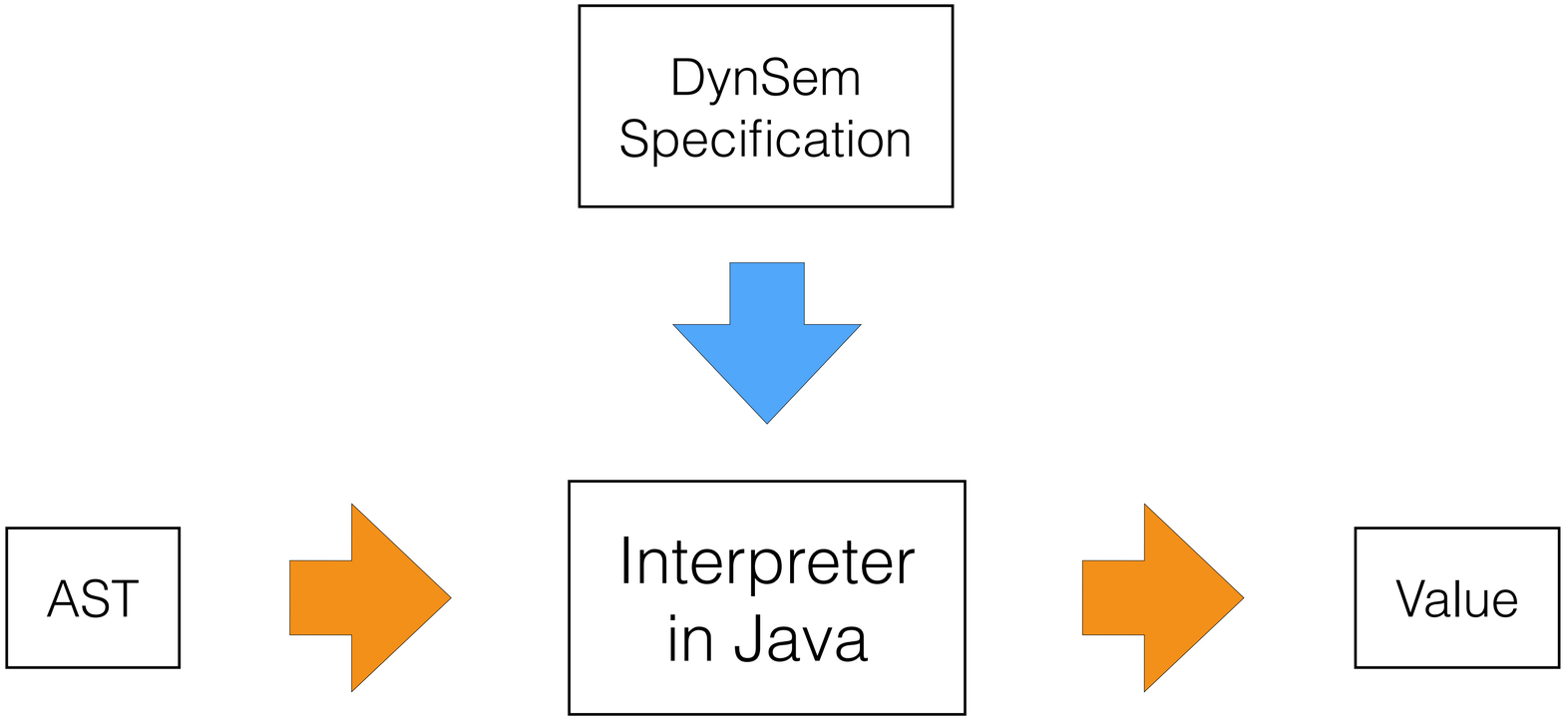
prettyprint.aterm

```
Mod(
  Let(
    [ TypeDecs(
        [ TypeDec(
            "tree"
          , RecordTy(
              [ Field("key", Tid("string"))
              , Field("left", Tid("tree"))
              , Field("right", Tid("tree"))
              ]
            )
          )
        ]
      )
    , FunDecs(
        [ FunDec(
            "prettyprint"
          , [FArg("tree", Tid("tree"))]
          , Tid("string")
          , Let(
              [ VarDecNoType("output", String("\"
              , FunDecs(
                  [ ProcDec(
                      "write"
                    , [FArg("s", Tid("string"))]
```

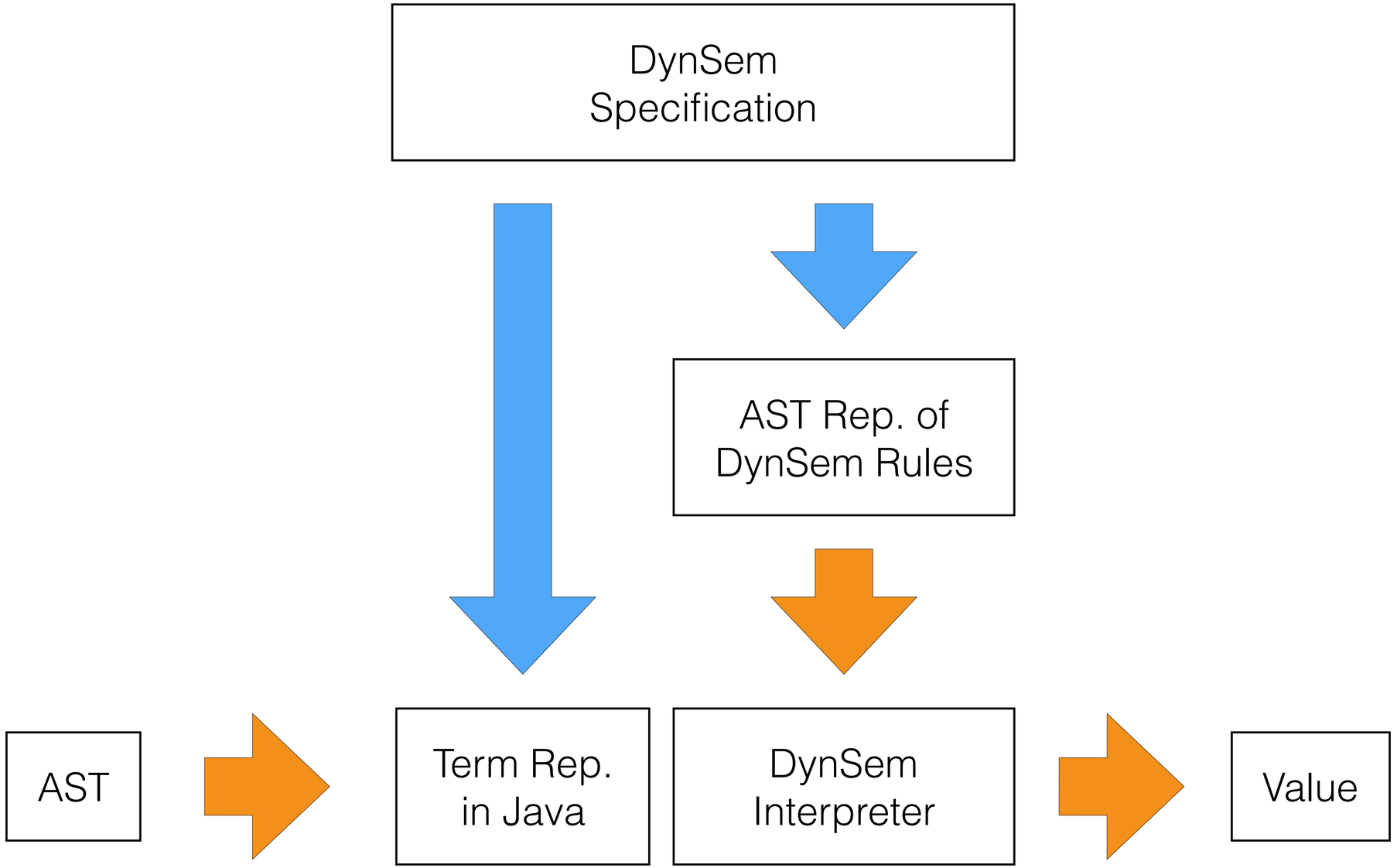
https://github.com/MetaBorgCube/metaborg-tiger

Interpreter Generation

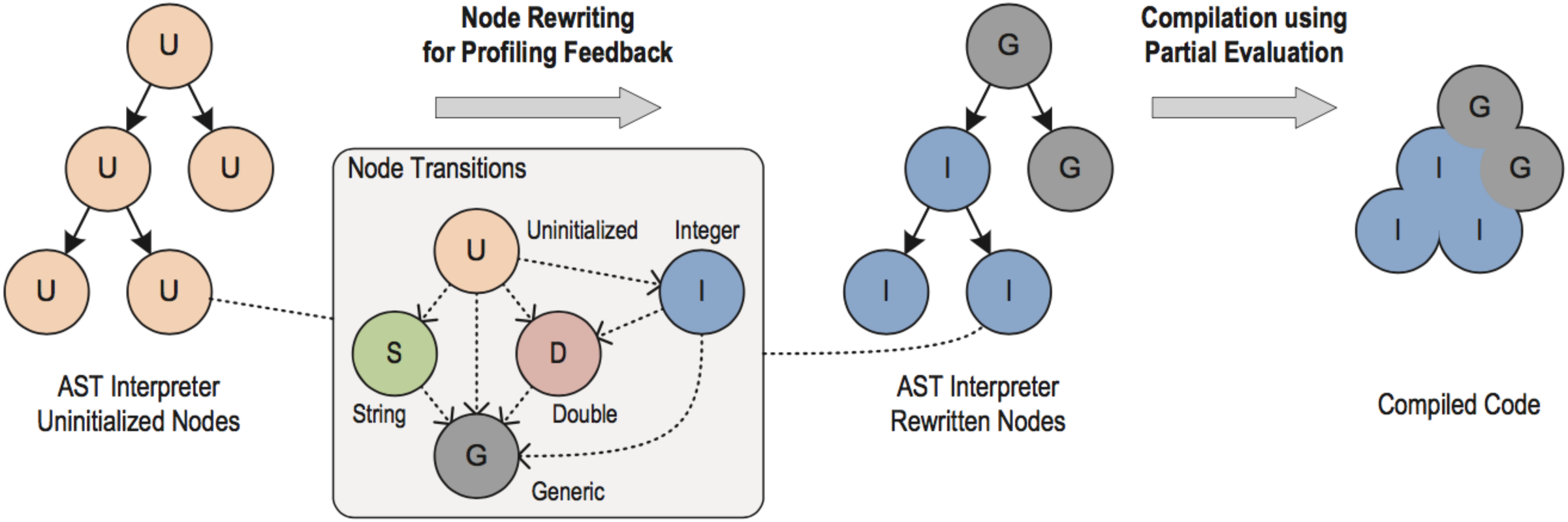
Generating AST Interpreter from Dynamic Semantics



DynSem Meta-Interpreter



Truffle: Partial Evaluation of AST Interpreters



Würthinger et al. One VM to rule them all. Onward! 2013

This paper describes the partial evaluation of the DynSem meta-interpreter with an object program.

The implementation makes use of Truffle a framework for run-time (online) partial evaluation.

Truffle gets most of it benefit from Graal, an implementation of the JVM with special support for partial evaluation.

ManLang 2018

https://doi.org/10.1145/3237009.3237018

Specializing a Meta-Interpreter

JIT Compilation of DynSem Specifications on the Graal VM

Vlad Vergu
TU Delft
The Netherlands
v.a.vergu@tudelft.nl

Eelco Visser
TU Delft
The Netherlands
visser@acm.org

ABSTRACT

DynSem is a domain-specific language for concise specification of the dynamic semantics of programming languages, aimed at rapid experimentation and evolution of language designs. DynSem specifications can be executed to interpret programs in the language under development. To enable fast turnaround during language development, we have developed a meta-interpreter for DynSem specifications, which requires minimal processing of the specification. In addition to fast development time, we also aim to achieve fast run times for interpreted programs.

In this paper we present the design of a meta-interpreter for DynSem and report on experiments with JIT compiling the application of the meta-interpreter on the Graal VM. By interpreting specifications directly, we have minimal compilation overhead. By specializing pattern matches, maintaining call-site dispatch chains and using native control-flow constructs we gain significant run-time performance. We evaluate the performance of the meta-interpreter when applied to the Tiger language specification running a set of common benchmark programs. Specialization enables the Graal VM to JIT compile the meta-interpreter giving speedups of up to factor 15 over running on the standard Oracle Java VM.

CCS CONCEPTS

• **Software and its engineering** → **Interpreters**; **Domain specific languages**; *Semantics*;

KEYWORDS

dynamic semantics, interpretation, JIT, run-time optimization

ACM Reference Format:
Vlad Vergu and Eelco Visser. 2018. Specializing a Meta-Interpreter: JIT Compilation of DynSem Specifications on the Graal VM. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3237009.3237018

1 INTRODUCTION

The dynamic semantics of a programming language defines the run time execution behavior of programs in the language. Ideally, the design of a programming language starts with the specification of its dynamic semantics to provide a high-level readable and unambiguous definition. However, understanding the design of a programming language also requires experimentation by actually running programs. Therefore, this ideal route is rarely taken, but language designs are embodied in the implementation of interpreters or compilers instead.

We have previously designed DynSem [33], a high-level meta-DSL for dynamic semantics specifications of programming languages, with the aim of supporting readable *and* executable specification. It supports the definition of modular and concise semantics by means of reduction rules with implicit propagation of contextual information. DynSem's executable semantics entails that specifications can be used to interpret object language programs.

In our early prototypes, DynSem specifications were compiled to an interpreter. The process of generating a Java implementation of an interpreter and compiling that generated code caused long turnaround times during language prototyping. In order to support rapid prototyping with short turnaround times, we turned to interpreting specifications directly instead of compiling them. A DynSem interpreter is a *meta-interpreter* since the programs it interprets are themselves interpreters. Figure 1 depicts the high-level architecture of the DynSem meta-interpreter. First, a DynSem specification is desugared (explicated) to make implicit passing of semantic components explicit. The resulting specification in DynSem Core is then loaded into the meta-interpreter together with the AST of the interpreted object program. The interpreter consumes the program as input enacting the specification. This produces the desired result of a short turnaround time for experimenting with dynamic semantics specifications.

Meta-interpretation reduces the turnaround time at the expense of execution performance. At run time there are two interpreter layers operating (the meta-language interpreter and the object-language interpreter) which introduces substantial overhead. While we envision DynSem as a convenient way to prototype the dynamic semantics of programming languages, ultimately we also envision it as a convenient way to bridge the gap between the prototyping and production phases of a programming language's lifecycle. Thus, we not only want an interpreter fast, but we also want a fast interpreter, which raises the question: Can we achieve fast object-language interpreters by optimizing the meta-interpretation of dynamic semantics specifications?

Direct vanilla interpreters are in general slow to begin with, even when they are implemented in a host language that is JIT-ed. This is because the host JIT is unable to see patterns in the object language and to meaningfully optimize the interpreter. The task of optimizing an interpreter has traditionally been long and

ManLang'18, September 12–14, 2018, Linz, Austria

ACM ISBN 978-1-4503-6424-9/18/09…$15.00
https://doi.org/10.1145/3237009.3237018

Partial Evaluation

Partial Evaluation = Program Specialization

Partial evaluation

=

Specializing a program to it's static (known) inputs

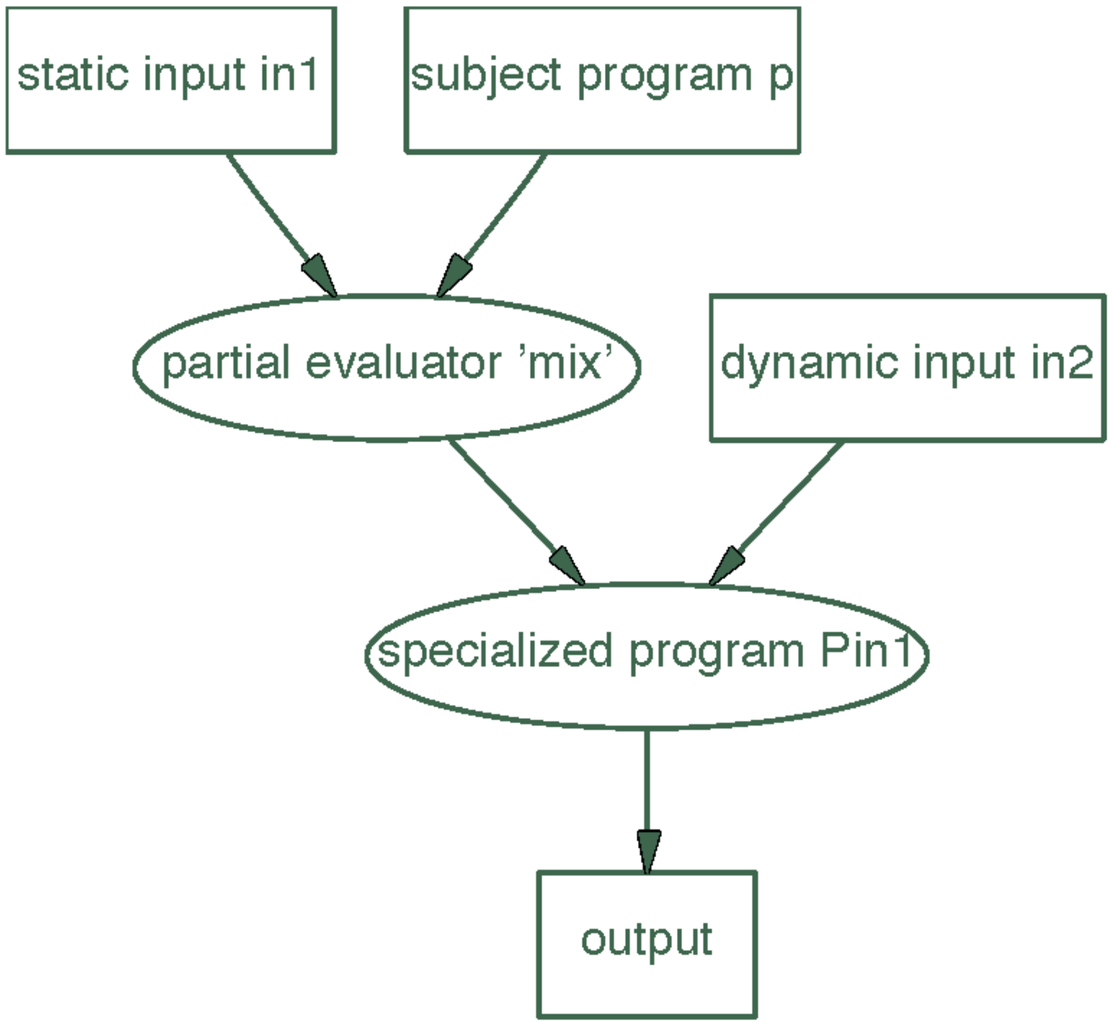
Source: Chapter 1 of Neil Jones, Carsten K. Gomar and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*

Kleene's S-M-N Theorem

There is a primitive recursive function σ such that

$$f(x, y) = \sigma(f, x)(y)$$

Partial Evaluator (Mix)



Example

```
function power(x : int, n : int) : int =
  if n = 0 then 1
  else if even(n) then square(power(x, n/2))
  else (x * power(x, n - 1))
```

Tiger

$\Downarrow$ `n = 5`

Tiger

```
function power5(x : int) : int =
  x * square(square(x))
```

precompute all expressions involving `n`
unfold recursive call to `power`
reduce `x*1` to `x`

Techniques

- *Symbolic computation*
  - compute with expressions involving variables (not only values)
- *Unfolding*
  - replace call with instantiated body of function
- *Program point specialization*
  - create new version of function, specialized to some arguments

  = definition + folding

Techniques

- *Definition*
  - introduce a new definition, or
  - extend an existing definition
- *Folding*
  - replace an expression with a function call
- *Memoization*
  - store the result of some computation
  - reproduce the result when computation is needed again

Terminology and Notation

- $\mathtt{p}$: program
- $[\![\mathtt{p}]\!]_{\mathtt{L}}$: meaning of program $\mathtt{p}$ in language $\mathtt{L}$
- $\mathtt{L}$: implementation language
- $\mathtt{S}$: source language
- $\mathtt{T}$: target language
- $[\![\_]\!]_{\mathtt{L}}$ has type $D \to D^* \to D$

$$\mathtt{output} = [\![\mathtt{p}]\!]_{\mathtt{L}}\ [\mathtt{in}_1, \mathtt{in}_2, ..., \mathtt{in}_n]$$

Equational Definition

one stage computation

$$\mathtt{out} = [\![\mathtt{p}]\!]\ [\mathtt{in1}, \mathtt{in2}]$$

two stage computation

$$\mathtt{p}_{\mathtt{in1}} = [\![\mathtt{mix}]\!]\ [\mathtt{p}, \mathtt{in1}]$$

$$\mathtt{out} = [\![\mathtt{p}_{\mathtt{in1}}]\!]\ \mathtt{in2}$$

equational definition of `mix`

$$[\![\mathtt{p}]\!]\ [\mathtt{in1}, \mathtt{in2}] = [\![[\![\mathtt{mix}]\!]\ [\mathtt{p}, \mathtt{in1}]]\!]\ \mathtt{in2}$$

different source, target, and implementation languages

$$[\![\mathtt{p}]\!]_{\mathtt{S}}\ [\mathtt{in1}, \mathtt{in2}] = [\![[\![\mathtt{mix}]\!]_{\mathtt{L}}\ [\mathtt{p}, \mathtt{in1}]]\!]_{\mathtt{T}}\ \mathtt{in2}$$

Why Partial Evaluation?

- Speedup
  - $\mathtt{t_p}(\mathtt{d}_1, ..., \mathtt{d}_n)$: time to compute $[\![p]\!]_{\mathtt{L}}\ [\mathtt{d}_1, ..., \mathtt{d}_n]$
  - $\mathtt{t_p}(\mathtt{in1}, \mathtt{in2})$: time to run program
  - $\mathtt{t_{mix}}(\mathtt{p}, \mathtt{in1})$: time to specialize p to in1
  - $\mathtt{t}_{\mathtt{p}_{\mathtt{in1}}}(\mathtt{in2})$: time to run specialized program
  - specialized program is faster (many applications to in1)

$$\mathtt{t}_{\mathtt{p}_{\mathtt{in1}}}(\mathtt{in2}) < \mathtt{t_p}(\mathtt{in1}, \mathtt{in2})$$

  - specialization and running is faster (one application to in1)

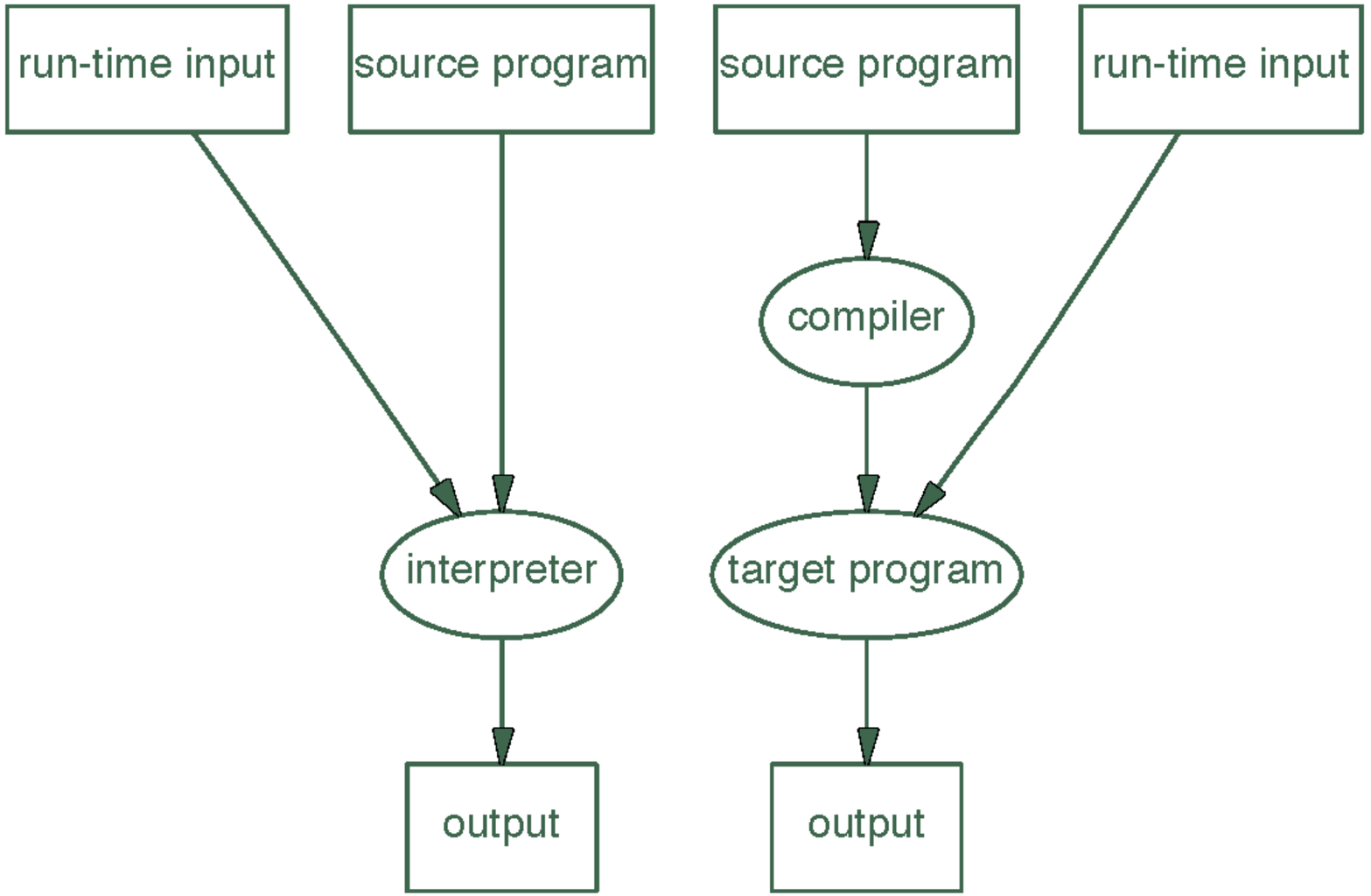
$$\mathtt{t_{mix}}(\mathtt{p}, \mathtt{in1}) + \mathtt{t}_{\mathtt{p}_{\mathtt{in1}}}(\mathtt{in2}) < \mathtt{t_p}(\mathtt{in1}, \mathtt{in2})$$

Why Partial Evaluation?

- Efficient *and* modular solution
  - *program adapted to one problem*: efficient, but difficult to maintain
  - *modular/generic program*: easy to maintain and produce different instance, but inefficient
  - *partial evaluation*: specialize generic program to specific instance

How is this relevant for compilers and interpreters?

Staged Computation



Interpreters and Compilers

interpreter `int` executes a source program

$$\begin{aligned} \texttt{output} &= [\![\texttt{source}]\!]_{\texttt{S}}\ [\texttt{in}_1, ..., \texttt{in}_n] \\ &= [\![\texttt{int}]\!]_{\texttt{L}}\ [\texttt{source}, \texttt{in}_1, ..., \texttt{in}_n] \end{aligned}$$

compiler generates object program from `source`

$$\texttt{target} = [\![\texttt{compiler}]\!]_{\texttt{L}}\ [\texttt{source}]$$

running `target`

$$\begin{aligned} \texttt{output} &= [\![\texttt{source}]\!]_{\texttt{S}}\ [\texttt{in}_1, ..., \texttt{in}_n] \\ &= [\![\texttt{target}]\!]_{\texttt{T}}\ [\texttt{in}_1, ..., \texttt{in}_n] \end{aligned}$$

Parser Generators

generating a parser

$$\texttt{parser}_{\texttt{grammar}} = [\![\texttt{parsegen}]\!]_{\texttt{L}}\ [\texttt{grammar}]$$

parsing a string

$$\texttt{parsetree} = [\![\texttt{parser}_{\texttt{grammar}}]\!]_{\texttt{L}}\ [\texttt{string}]$$

general parsers

$$\texttt{parsetree} = [\![\texttt{genparser}]\!]_{\texttt{L}}\ [\texttt{grammar}, \texttt{string}]$$

Generalization: Executable Specifications

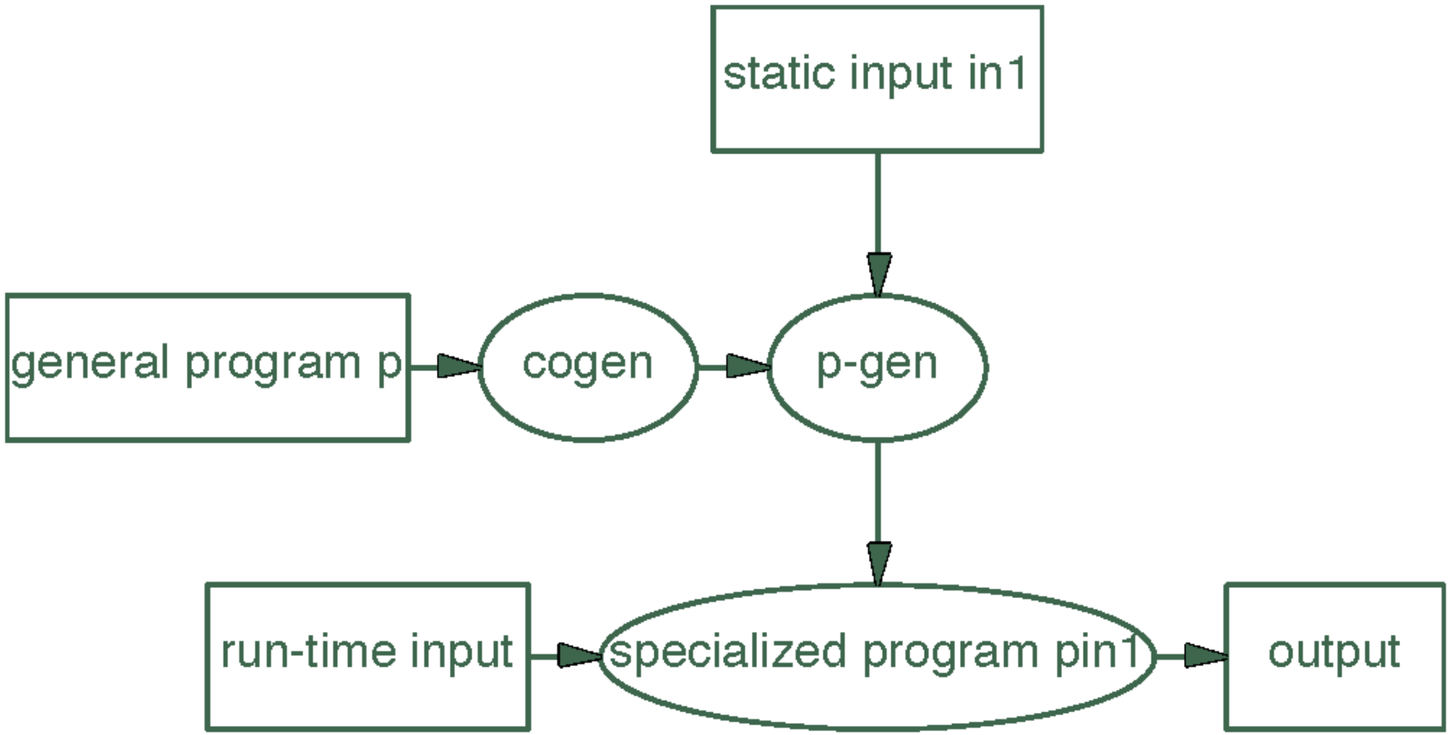
specification executer `specexec` 'runs' specification `spec`

$$[\![\texttt{specexec}]\!]\ [\texttt{spec}, \texttt{in}_1, ..., \texttt{in}_n]$$

`specexec` is highly generic program

examples of such executers: `int`, `genparser`

Program Generator Generation



$$\texttt{compiler} = [\![\texttt{cogen}]\!]\ \texttt{int}$$

$$\texttt{parsegen} = [\![\texttt{cogen}]\!]\ \texttt{genparser}$$

Compiling by Partial Evaluation

First Futamura Projection (1971): specialize interpreter to source

$$\texttt{target} = [\![\texttt{mix}]\!]\ [\texttt{int}, \texttt{source}]$$

Compiling by partial evaluation yields correct target program:

$$\begin{aligned}
\texttt{out} &= [\![\texttt{source}]\!]_{\texttt{S}}\ \texttt{input} \\
&= [\![\texttt{int}]\!]\ [\texttt{source}, \texttt{input}] \\
&= [\![[\![\texttt{mix}]\!]\ [\texttt{int}, \texttt{source}]]\!]\ \texttt{input} \\
&= [\![\texttt{target}]\!]\ \texttt{input}
\end{aligned}$$

- target is in output language of `mix`
- specialization removes overhead of inspecting source

Compiler Generation

Second Futamura Projection

$$\texttt{compiler} = [\![\texttt{mix}]\!]\ [\texttt{mix}, \texttt{int}]$$

Correctness

$$\begin{aligned} \texttt{target} &= [\![\texttt{mix}]\!]\ [\texttt{int}, \texttt{source}] \\ &= [\![[\![\texttt{mix}]\!]\ [\texttt{mix}, \texttt{int}]]\!]\ \texttt{source} \\ &= [\![\texttt{compiler}]\!]\ \texttt{source} \end{aligned}$$

Compiler Generator Generation

Third Futamura Projection

$$\texttt{cogen} = [\![\texttt{mix}]\!]\ [\texttt{mix}, \texttt{mix}]$$

Correctness

$$\begin{aligned}
\texttt{compiler} &= [\![\texttt{mix}]\!]\ [\texttt{mix}, \texttt{int}] \\
&= [\![[\![\texttt{mix}]\!]\ [\texttt{mix}, \texttt{mix}]]\!]\ \texttt{int} \\
&= [\![\texttt{cogen}]\!]\ \texttt{source}
\end{aligned}$$

Partial Evaluation: Summary

Instance-specific program

- Can make very efficient
- Hard to maintain (performance interventions tangled with implementation)
- Hard to reuse for variants

Generic program

- Parametric in instances
- Easy to maintain
- Easy to reuse for variants
- Not efficient: overhead of parametricity

Partial evaluation

- Specialize generic program to specific instances

Some Research Projects

Specializing a meta-interpreter

- Meta-interpreter: DynSem specifications executed using interpreter
- Partial evaluation: remove two stages of interpretation
- Encouraging results using Truffle/Graal
- How close to native speed can we get?

Generate byte code compiler

- Input: DynSem specification
- Output: compiler from source to (custom) byte code
- Output: byte code interpreter

Further Study

Partial evaluation

- Offline vs online partial evaluation
- Binding-time analysis

Multi-stage programming

- Explicit binding time annotations

Just-in-time (JIT) compilation

- Interpreter selectively compiles code (fragments) to machine code

Techniques for partial evaluation

- Meta-tracing
- Run-time specialisation (Truffle/Graal)