Functions, Calling Conventions, and Code Generation Mechanics

Eelco Visser

TU Delft

CS4200 | Compiler Construction | December 3, 2020

This Lecture

Functions

- calling conventions

Context-sensitive transformations

- dynamic rewrite rules

Code Generation Mechanics

- properties of code generators

Functions

Functions in ChocoPy

function name

local variables

return to caller

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def caller():
  d : int = 0
  d = callee(345, 4357, 235)
```

formal parameters

call function

actual parameters

Operational Semantics: Invoke

$$\frac{\begin{array}{l} S_0(E(f)) = (x_1, \ldots, x_n, y_1 = e'_1, \ldots, y_k = e'_k, b_{body}, E_f) \\ n, k \geq 0 \\ G, E, S_0 \vdash e_1 : v_1, S_1, \_ \\ \vdots \\ G, E, S_{n-1} \vdash e_n : v_n, S_n, \_ \\ l_{x1}, \ldots, l_{xn}, l_{y1}, \ldots, l_{yk} = newloc(S_n, n+k) \\ E' = E_f[l_{x1}/x_1] \ldots [l_{xn}/x_n][l_{y1}/y_1] \ldots [l_{yk}/y_k] \\ G, E', S_n \vdash e'_1 : v'_1, S_n, \_ \\ \vdots \\ G, E', S_n \vdash e'_k : v'_k, S_n, \_ \\ S_{n+1} = S_n[v_1/l_{x1}] \ldots [v_n/l_{xn}][v'_1/l_{y1}] \ldots [v'_k/l_{yk}] \\ G, E', S_{n+1} \vdash b_{body} : \_, S_{n+2}, R \\ R' = \begin{cases} None, & \text{if } R \text{ is } \_ \\ R, & \text{otherwise} \end{cases} \end{array}}{G, E, S_0 \vdash f(e_1, \ldots, e_n) : R', S_{n+2}, \_} \quad [\text{INVOKE}]$$

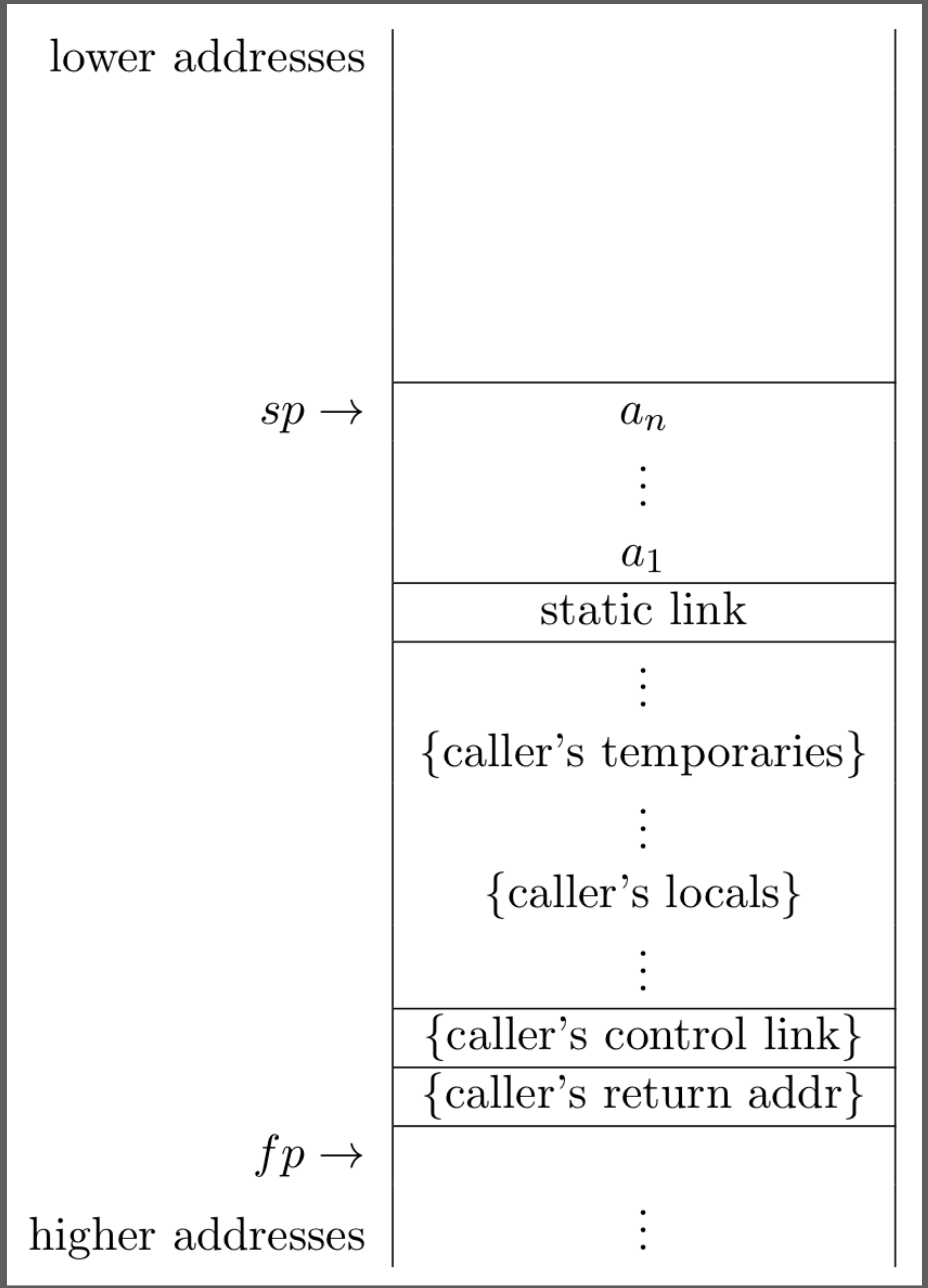
Operational Semantics: Invoke

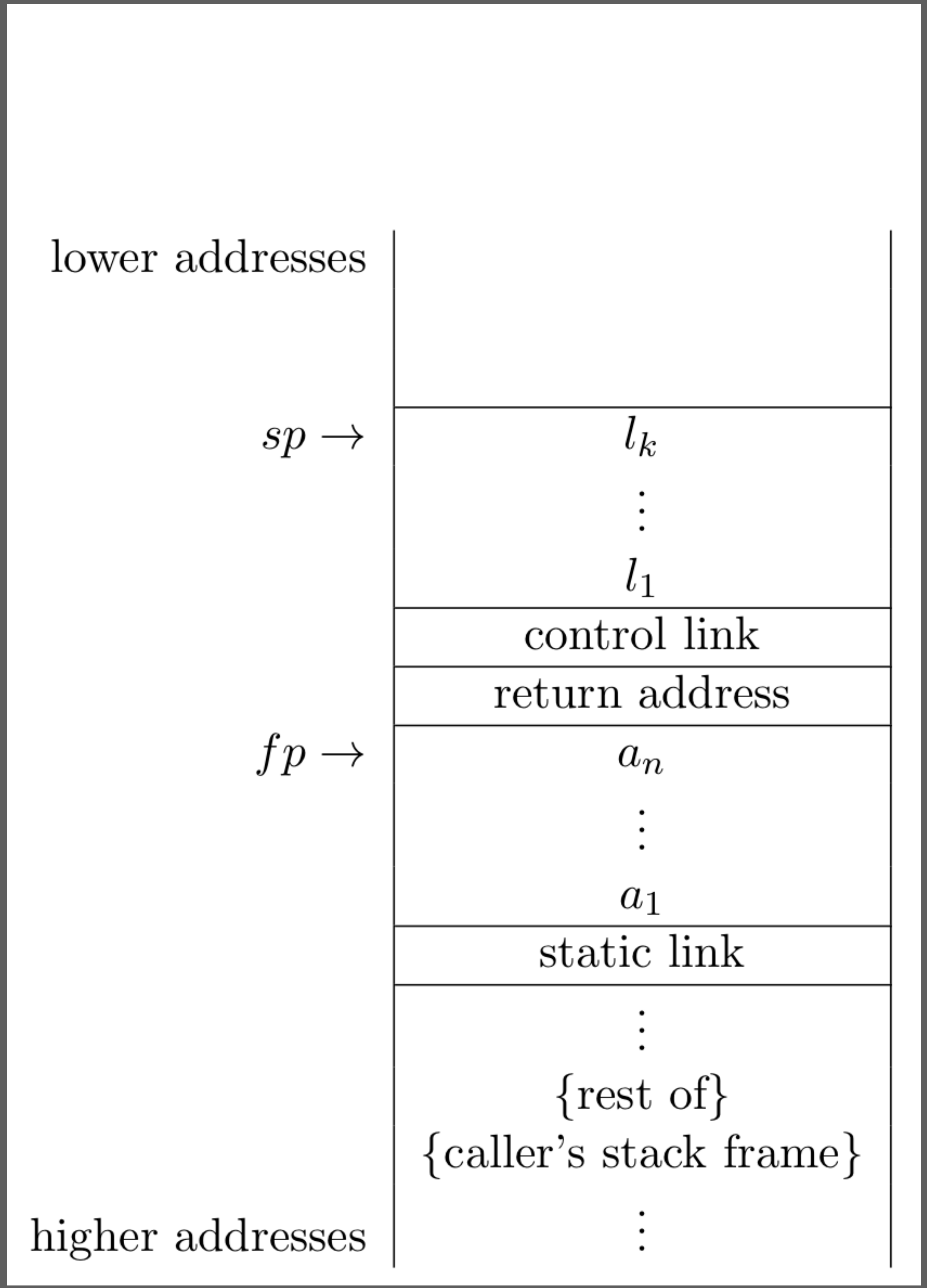
$$\frac{\begin{array}{l} g_1, \ldots, g_L \text{ are the variables explicitly declared as global in } f \\ y_1 = e_1, \ldots, y_k = e_k \text{ are the local variables and nested functions defined in } f \\ E_f = E[G(g_1)/g_1] \ldots [G(g_L)/g_L] \\ v = (x_1, \ldots, x_n, y_1 = e_1, \ldots, y_k = e_k, b_{body}, E_f) \end{array}}{G, E, S \vdash \texttt{def } f(x_1 : T_1, \ldots, x_n : T_n) \ [\![\texttt{->} \ T_0]\!]^? : b : v, S, \_} \quad [\text{FUNC-METHOD-DEF}]$$

Activation Records

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def caller():
  d : int = 0
  d = callee(345, 4357, 235)
```





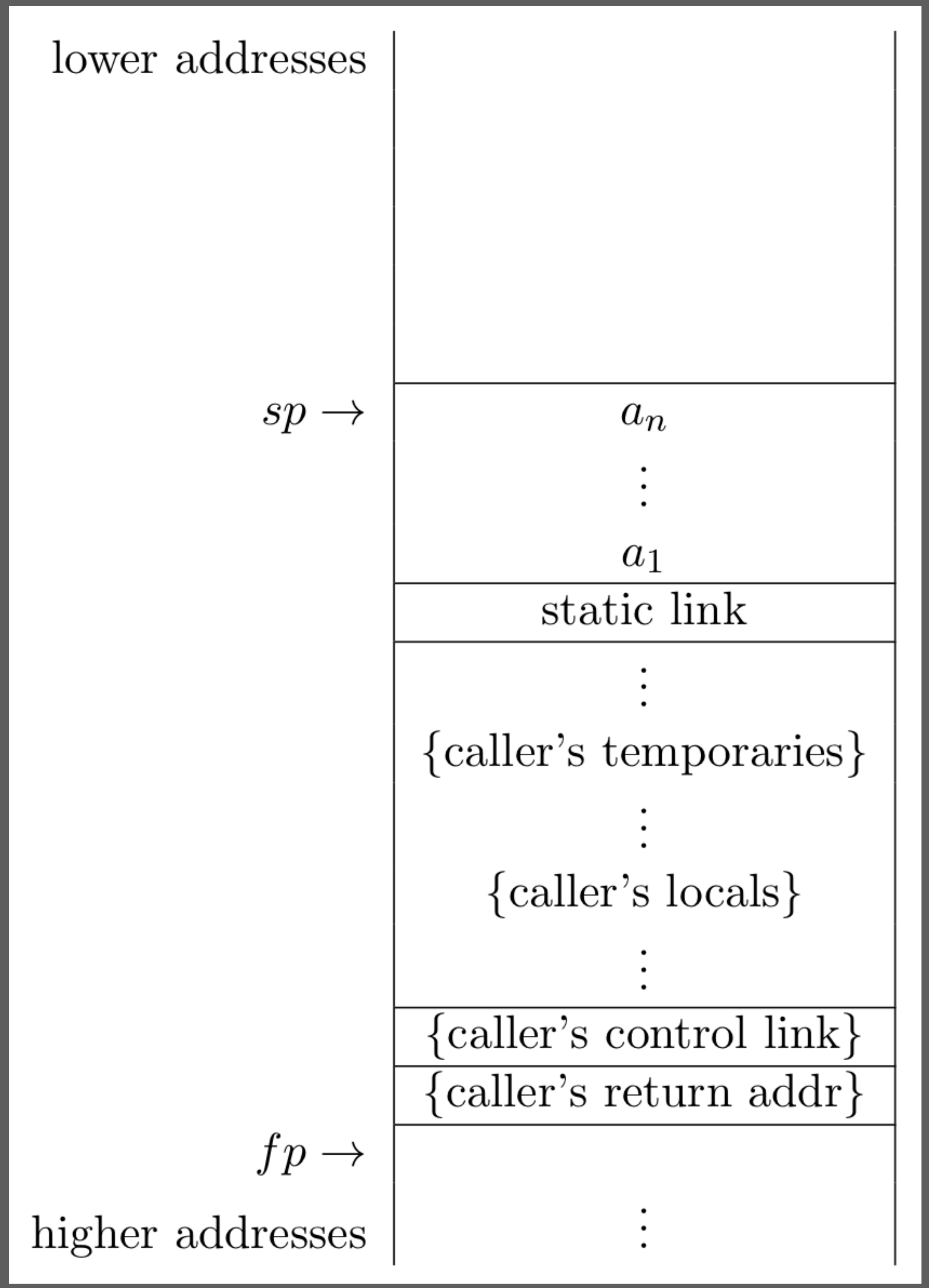
before/after invocation

during callee's execution

Calling Convention: Caller

```python
def callee(x : int, y : int, z: int) -> int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def caller():
  d : int = 0
  d = callee(345, 4357, 235)
```

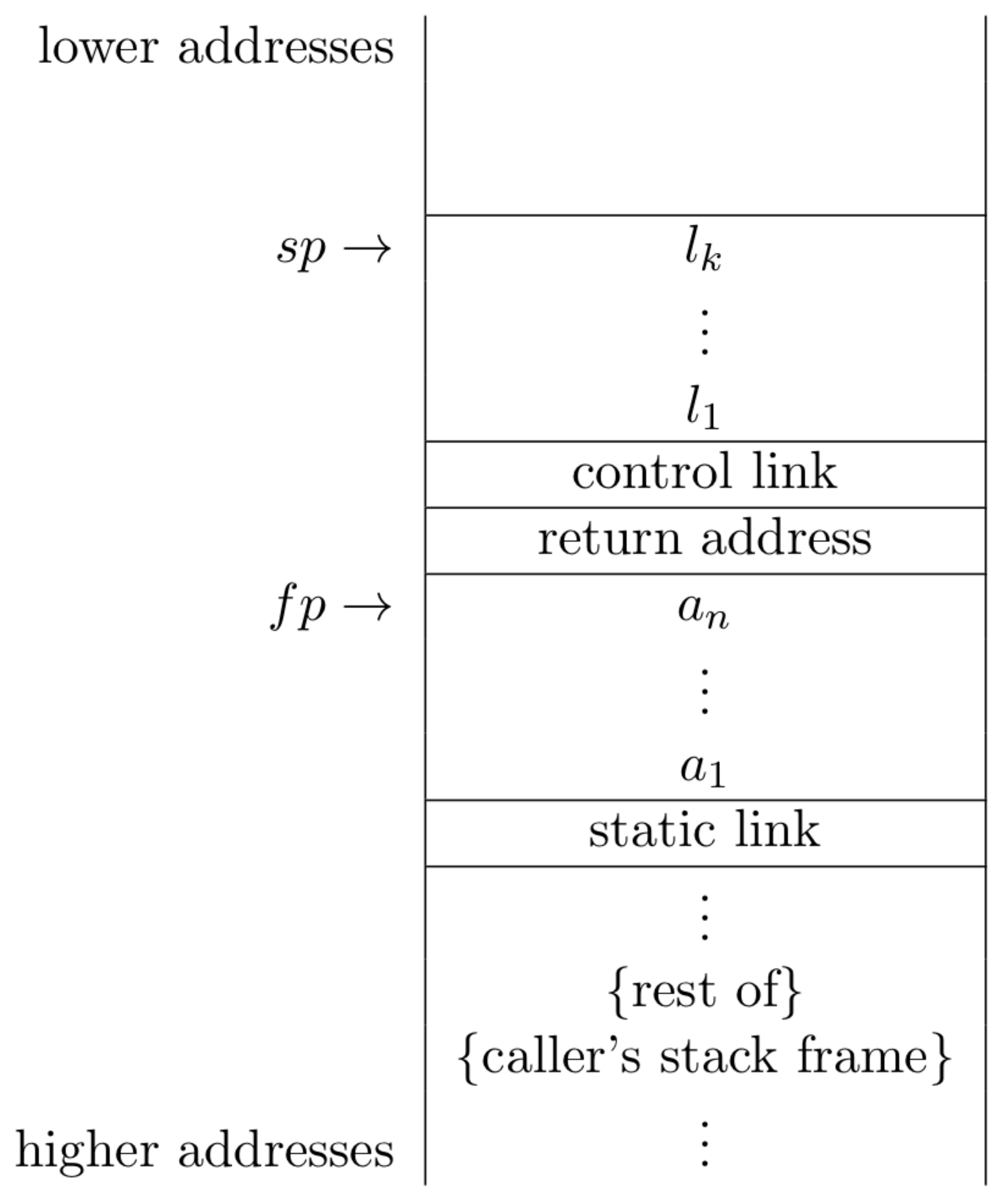



```
.globl $caller
$caller:
addi    sp, sp, -@caller.size
sw      ra, @caller.size-4(sp)
sw      fp, @caller.size-8(sp)
sw      fp, @caller.size(sp)
li      a0, 0                # initialize local variable d
sw      a0, -12(fp)
sw      sp, -12(sp)          # reserve space for arguments
li      a0, 345              # evaluate first argument
sw      a0, 12(sp)           # push on stack
li      a0, 4357             # evaluate second argument
sw      a0, 8(sp)            # push on stack
li      a0, 235              # evaluate third argument
sw      a0, 4(sp)            # push on stack
jal     $callee              # call function
sw      sp, 12(sp)           # clean up stack
sw      a0, -12(fp)          # return value in a0
j       label_48
mv      a0, zero
j       label_48
label_48:
.equiv @caller.size, 12
lw      ra, -4(fp)
lw      fp, -8(fp)
addi    sp, sp, @caller.size
jr      ra
```

Calling Convention: Callee

```python
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def caller():
  d : int = 0
  d = callee(345, 4357, 235)
```




```asm
.globl $callee
$callee:
addi    sp, sp, -@callee.size  # reserve space for stack frame
sw      ra, @callee.size-4(sp) # save return address
sw      fp, @callee.size-8(sp) # save control link (fp)
sw      fp, @callee.size(sp)   # new fp is at old SP
li      a0, 1                  # initialize local variable a
sw      a0, -16(fp)
li      a0, 2                  # initialize local variable b
sw      a0, -12(fp)
lw      a0, 8(fp)              # load argument x
lw      t0, 4(fp)              # load argument y
add     a0, a0, t0             # x + y
lw      t0, 0(fp)              # load argument z
add     a0, a0, t0             # (x + y) + z
lw      t0, -16(fp)            # load local variable a
add     a0, a0, t0             # (x + y + z) + a
lw      t0, -12(fp)            # load local variable b
add     a0, a0, t0             # (x + y + z + a) + b
j       label_47
mv      a0, zero
j       label_47
label_47:
.equiv @callee.size, 16
lw      ra, -4(fp)             # restore return address
lw      fp, -8(fp)             # restore frame pointer
addi    sp, sp, @callee.size   # restore stack pointer
jr      ra                     # return to caller
```

Calling a Function in Function Call Argument

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller():
  d : int = 0
  d = callee(345 + 81 + inc(13), 4357, 235)
```

problem: callee overwrites registers for temporaries

Calling a Function in Function Call Argument

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller():
  d : int = 0
  d = callee(345 + 81 + inc(13), 4357, 235)
```

```
def callee(x : int, y : int, z : int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller( ) :
  d : int = 0
  temp_2 : int = 0
  temp_2 = inc(13)
  d = callee(345 + 81 + temp_2, 4357, 235)
```

problem: callee overwrites registers for temporaries

solution: lift calls from call expressions
store result in local variable

Calling a Function in Function Call Argument

```python
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller():
  d : int = 0
  d = callee(345 + 81 + inc(13), 4357, 235)
```

problem: callee overwrites registers for temporaries

```python
def callee(x : int, y : int, z : int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller( ) :
  d : int = 0
  temp_2 : int = 0
  temp_2 = inc(13)
  d = callee(345 + 81 + temp_2, 4357, 235)
```

solution: lift calls from call expressions
store result in local variable

```
.globl $caller
$caller:
addi    sp, sp, -@caller.size
sw      ra, @caller.size-4(sp)
sw      fp, @caller.size-8(sp)
sw      fp, @caller.size(sp)
li      a0, 0
sw      a0, -16(fp)    # init local variable d
li      a0, 0
sw      a0, -12(fp)    # init local variable temp_2
sw      sp, -4(sp)     # prepare for call to inc
li      a0, 13
sw      a0, 4(sp)
jal     $inc
sw      sp, 4(sp)      # clean up after call to inc
sw      a0, -12(fp)    # store return value of inc in temp_2
sw      sp, -12(sp)    # prepare for call to callee
li      a0, 345        # evaluate first argument
addi    a0, a0, 81
lw      t0, -12(fp)    # load local variable temp_2
add     a0, a0, t0
sw      a0, 12(sp)     # push first argument to stack
li      a0, 4357
sw      a0, 8(sp)      # push second argument to stack
li      a0, 235
sw      a0, 4(sp)      # push third argument to stack
jal     $callee
sw      sp, 12(sp)     # clean up stack after call to callee
sw      a0, -16(fp)    # return value of callee in local variable d
j       label_40
mv      a0, zero
j       label_40
label_40:
.equiv @caller.size, 16
lw      ra, -4(fp)
lw      fp, -8(fp)
addi    sp, sp, @caller.size
jr      ra
```

Calling a Function in Function Call Argument: Zooming In

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller():
  d : int = 0
  d = callee(345 + 81 + inc(13), 4357, 235)
```

problem: callee overwrites registers for temporaries

```
def callee(x : int, y : int, z : int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller( ) :
  d : int = 0
  temp_2 : int = 0
  temp_2 = inc(13)
  d = callee(345 + 81 + temp_2, 4357, 235)
```

solution: lift calls from call expressions
store result in local variable

```
.globl $caller
$caller:
addi    sp, sp, -@caller.size
sw      ra, @caller.size-4(sp)
sw      fp, @caller.size-8(sp)
sw      fp, @caller.size(sp)

li      a0, 0
sw      a0, -16(fp)    # init local variable d
li      a0, 0
sw      a0, -12(fp)    # init local variable temp_2

sw      sp, -4(sp)     # prepare for call to inc
li      a0, 13
sw      a0, 4(sp)
jal     $inc
sw      sp, 4(sp)      # clean up after call to inc
sw      a0, -12(fp)    # store return value of inc in temp_2

sw      sp, -12(sp)    # prepare for call to callee
li      a0, 345        # evaluate first argument
addi    a0, a0, 81
lw      t0, -12(fp)    # load local variable temp_2
add     a0, a0, t0
sw      a0, 12(sp)     # push first argument to stack
li      a0, 4357
sw      a0, 8(sp)      # push second argument to stack
li      a0, 235
sw      a0, 4(sp)      # push third argument to stack
jal     $callee
sw      sp, 12(sp)     # clean up stack after call to callee
sw      a0, -16(fp)    # return value of callee in local variable d
```

locals

call to inc

call to callee

Nested Functions

```
def callee(x : int, y : int, z : int) → int:
  def incZ(i : int) → int:
    return i + z
  b : int = 234
  b = incZ(x)
  return y + b

def caller( ) :
  d : int = 0
  d = callee(345 + 81 , 4357, 235)
```

Optimizations

Local variable initialization

– don’t, if it is always assigned to before use

Storing value on stack

– don’t, if it is immediately retrieved

Context-Sensitive Transformation with Scoped Dynamic Rewrite Rules

Counting Stack

```
rules

  stack-set(|n) =
    rules(Stack : () → n); !n

  stack-get =
    <Stack>() <+ !0

  stack-inc(|n) =
    stack-set(|<add>(<stack-get>, n))
```

define rewrite rule dynamically

invoke dynamic rewrite rule

```
{| Stack
 : stack-set(|0)
 ; <some-transformation> t ⇒ instrs
 ; size := <stack-get>
 |}
```

dynamic rule scope

forget dynamic rules added within scope

Keeping Track of Local Variables

```
rules

  var-offset-set(|x, n) =
    rules(VarOffset : x → n)

  var-offset-get :
    x → n
    with <VarOffset> x ⇒ n
```

define rewrite rule dynamically

invoke dynamic rewrite rule

```
rules

  fun-arg :
    TypedVar(x, t) → offset
    with var-offset-set(|x, <stack-get ⇒ offset>)
    with stack-inc(|4)

  exp-to-instrs-(|r, regs) :
    Var(x) → [Lw(r, <int-to-string>offset, "fp")]
    with <var-offset-get>x ⇒ offset
```

bind offset of formal parameter

lookup offset of formal parameter

Code Generation Mechanics

Code Generation Mechanics

Code generation

- Input: AST of source language program
  - with name and type annotations
- Output: machine instructions

Mechanics

- What techniques are available to define translation?
- What are the advantages and disadvantages of these techniques?
- To what extent do these techniques help with verification?

Code Generation by String Manipulation

Printing Strings as Side Effect

```
to-jbc = ?Nil()   ; <printstring> "aconst_null\n"
to-jbc = ?NoVal() ; <printstring> "nop\n"
to-jbc = ?Seq(es) ; <list-loop(to-jbc)> es

to-jbc =
   ?Int(i);
   <printstring> "ldc ";
   <printstring> i;
   <printstring> "\n"

to-jbc = ?Bop(op, e1, e2) ; <to-jbc> e1 ; <to-jbc> e2 ; <to-jbc> op

to-jbc = ?PLUS()  ; <printstring> "iadd\n"
to-jbc = ?MINUS() ; <printstring> "isub\n"
to-jbc = ?MUL()   ; <printstring> "imul\n"
to-jbc = ?DIV()   ; <printstring> "idiv\n"
```

String Concatenation

```
to-jbc: Nil()   -> "aconst_null\n"
to-jbc: NoVal() -> "nop\n"
to-jbc: Seq(es) -> <concat-strings> <map(to-jbc)> es

to-jbc: Int(i) -> <concat-strings> ["ldc ", i, "\n"]

to-jbc: Bop(op, e1, e2) -> <concat-strings> [ <to-jbc> e1,
                                              <to-jbc> e2,
                                              <to-jbc> op ]


to-jbc: PLUS()  -> "iadd\n"
to-jbc: MINUS() -> "isub\n"
to-jbc: MUL()   -> "imul\n"
to-jbc: DIV()   -> "idiv\n"
```

String Interpolation

```
to-jbc: Nil()   -> $[aconst_null]
to-jbc: NoVal() -> $[nop]
to-jbc: Seq(es) -> <map-to-jbc> es

map-to-jbc: [] -> $[]
map-to-jbc: [h|t] ->
   $[[<to-jbc> h]
     [<map-to-jbc> t]]

to-jbc: Int(i) -> $[ldc [i]]
to-jbc: Bop(op, e1, e2) ->
 $[[<to-jbc> e1]
   [<to-jbc> e2]
   [<to-jbc> op]]

to-jbc: PLUS()  -> $[iadd]
to-jbc: MINUS() -> $[isub]
to-jbc: MUL()   -> $[imul]
to-jbc: DIV()   -> $[idiv]
```

Summary: Code Generation by String Manipulation

Printing strings

- Generated code depends on order of traversal of the AST
- Explicit layout (whitespace) management
- Verbose quotation and anti-quotation
- Escaping meta-variables
- Easy to make syntax errors
- Output needs to be parsed for further processing

String concatenation

- Makes generation order independent

String interpolation (templates)

- Makes quotation and anti-quotation more concise
- Layout (whitespace) from template layout

Correctness of String-Based Code Generators

All bets are off

- Only guarantee is that you get some text
- String interpolation may help with producing readable code
- Very easy to make even trivial syntactic errors

Verification

- Use target code checker for verification
- No input independent guarantees

Code Generation by Term Transformation

Code Generation by Transformation

AST to AST translation

- input: source language AST
- output: target language AST

Defined using term rewrite rules

- Recognise AST pattern for language construct
- Recursively translate sub-terms
- Compose results with target code schema for language construct

Intermediate representation (IR)

Code Generation by Transformation: Example

```
to-jbc : Exp -> List(Instruction)
```

```
to-jbc: Nil()   -> [ ACONST_NULL() ]
to-jbc: NoVal() -> [ NOP() ]
to-jbc: Seq(es) -> <mapconcat(to-jbc)> es

to-jbc: Int(i)    -> [ LDC(Int(i)) ]
to-jbc: String(s) -> [ LDC(String(s)) ]

to-jbc: Bop(op, e1, e2) -> <mapconcat(to-jbc)> [ e1, e2, op ]

to-jbc: PLUS()  -> [ IADD() ]
to-jbc: MINUS() -> [ ISUB() ]
to-jbc: MUL()   -> [ IMUL() ]
to-jbc: DIV()   -> [ IDIV() ]

to-jbc: Assign(lhs, e) -> <concat> [ <to-jbc> e, <lhs-to-jbc> lhs ]

to-jbc:     Var(x) -> [ ILOAD(x) ]  where <type-of> Var(x) => INT()
to-jbc:     Var(x) -> [ ALOAD(x) ]  where <type-of> Var(x) => STRING()
lhs-to-jbc: Var(x) -> [ ISTORE(x) ] where <type-of> Var(x) => INT()
lhs-to-jbc: Var(x) -> [ ASTORE(x) ] where <type-of> Var(x) => STRING()
```

Code Generation by Transformation: Example

```
to-jbc:
   IfThenElse(e1, e2, e3) -> <concat>   [ <to-jbc> e1
                                        , [ IFEQ(LabelRef(else)) ]
                                        , <to-jbc> e2
                                        , [ GOTO(LabelRef(end)), Label(else) ]
                                        , <to-jbc> e3
                                        , [ Label(end) ]
                                        ]
   where <newname> "else" => else
   where <newname> "end"  => end

to-jbc:
   While(e1, e2) -> <concat> [ [ GOTO(LabelRef(check)), Label(body) ]
                             , <to-jbc> e2
                             , [ Label(check) ]
                             , <to-jbc> e1
                             , [ IFNE(LabelRef(body)) ]
                             ]
   where <newname> "test" => check
   where <newname> "body" => body
```

Code Generation by Transformation

Compiler component composition

- AST output can be consumed by compatible AST transformations

Example compilation pipeline

- Parse source language text => source language AST
- Desugar => source language AST
- Type-check => annotated source language AST
- Translate => target language AST
- Optimize => target language AST
- Pretty-print => target language text

Easy to extend with new components

Guaranteeing Syntactically Correct Target Code

Syntactically Correct Target Code

Property: Syntactically correct target code

- Guarantee that generated code parses

Type correct AST = syntactically correct code

- AST types represent syntactic categories
  - `Plus: Exp * Exp -> Exp`
- Type check translation patterns

Language support

- Any programming language with a static type system
- And support for algebraic data types

Note: lexical syntax

Type Checking Transformation Rules

```
module Tiger-Condensed
signature
  constructors
    Var         : Id -> Var
    String      : StrConst -> Exp
    Seq         : List(Exp) -> Exp
    Call        : Var * List(Exp) -> Exp
    Plus        : Exp * Exp -> Exp
    Minus       : Exp * Exp -> Exp
    Assign      : Var * Exp -> Exp
    If          : Exp * Exp * Exp -> Exp
    Let         : List(Dec) * List(Exp) -> Exp
    VarDec      : Id * TypeAn * Exp -> Dec
    FunctionDec : List(FunDec) -> Dec
    FunDec      : Id * List(FArg) * TypeAn * Exp -> FunDec
    FArg        : Id * TypeAn -> FArg
    NoTp        : TypeAn
    Tp          : TypeId -> TypeAn
```

```
module Tiger-TraceAll
imports Tiger-Typed lib Tiger-Simplify
strategies
  instrument = topdown(try(TraceProcedure + TraceFunction));
               IntroducePrinters; simplify
rules
  TraceProcedure :
    FunDec(f, xs, NoTp, e) ->
    FunDec(f, xs, NoTp,
           Seq([Call(Var("enterfun"),[String(f)]), e,
                Call(Var("exitfun"),[String(f)])]))
  TraceFunction :
    FunDec(f, xs, Tp(tid), e) ->
    FunDec(f, xs, Tp(tid),
           Seq([Call(Var("enterfun"),[String(f)]),
                Let([VarDec(x,Tp(tid),NilExp)],
                    [Assign(Var(x), e),
                     Call(Var("exitfun"),[String(f)]),
                     Var(x)])]))
    where new => x
  IntroducePrinters :
    e -> /* omitted for brevity */
```

Type checking terms in rules guarantees *syntactic* correctness of generated code

Guaranteeing Syntactically Correct Target Code in Stratego?

:-(

Stratego

- Only checks arities of constructor applications, not types
- Transformation rules could be checked by the compiler
- Generic traversals make traditional type checking impossible

Research

- A static analysis for Stratego that guarantees syntactic correctness

Workaround

- Meta-programming with concrete object syntax

This paper defines a generic technique for embedding the concrete syntax of an object language into a meta-programming language.

Applied to Stratego as meta-language and Tiger as object language.

Combines two advantages
- guarantee syntactic correctness of match and build patterns
- make rules more readable

https://doi.org/10.1007/3-540-45821-2_19

Meta-programming with Concrete Object Syntax

Eelco Visser

Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands. http://www.cs.uu.nl/~visser, visser@acm.org

Abstract. Meta programs manipulate structured representations, i.e., abstract syntax trees, of programs. The conceptual distance between the concrete syntax meta-programmers use to reason about programs and the notation for abstract syntax manipulation provided by general purpose (meta-) programming languages is too great for many applications. In this paper it is shown how the syntax definition formalism SDF can be employed to fit *any* meta-programming language with concrete syntax notation for composing and analyzing object programs. As a case study, the addition of concrete syntax to the program transformation language Stratego is presented. The approach is then generalized to arbitrary meta-languages.

1 Introduction

Meta-programs analyze, generate, and transform object programs. In this process object programs are structured data. It is common practice to use abstract syntax trees rather than the textual representation of programs [10]. Abstract syntax trees are represented using the data structuring facilities of the meta-language: records (structs) in imperative languages (C), objects in object-oriented languages (C++, Java), algebraic data types in functional languages (ML, Haskell), and terms in term rewriting systems (Stratego).

Such representations allow the full capabilities of the meta-language to be applied in the implementation of meta-programs. In particular, when working with high-level languages that support symbolic manipulation by means of pattern matching (e.g., ML, Haskell) it is easy to compose and decompose abstract syntax trees. For meta-programs such as compilers, programming with abstract syntax is adequate; only small fragments, i.e., a few constructors per pattern, are manipulated at a time. Often, object programs are reduced to a core language that only contains the essential constructs. The abstract syntax can then be used as an intermediate language, such that multiple languages can be expressed in it, and meta-programs can be reused for several source languages.

However, there are many applications of meta-programming in which the use of abstract syntax is not satisfactory since the conceptual distance between the

D. Batory, C. Consel, and W. Taha (Eds.): GPCE 2002, LNCS 2487, pp. 299–315, 2002.

Concrete Object Syntax

```
module Tiger-TraceAll
imports Tiger-Typed lib Tiger-Simplify
strategies
  instrument = topdown(try(TraceProcedure + TraceFunction));
               IntroducePrinters; simplify
rules
  TraceProcedure :
    FunDec(f, xs, NoTp, e) ->
    FunDec(f, xs, NoTp,
           Seq([Call(Var("enterfun"),[String(f)]), e,
                Call(Var("exitfun"),[String(f)])]))
  TraceFunction :
    FunDec(f, xs, Tp(tid), e) ->
    FunDec(f, xs, Tp(tid),
           Seq([Call(Var("enterfun"),[String(f)]),
                Let([VarDec(x,Tp(tid),NilExp)],
                    [Assign(Var(x), e),
                     Call(Var("exitfun"),[String(f)]),
                     Var(x)])]))
    where new => x
  IntroducePrinters :
    e -> /* omitted for brevity */
```

Abstract syntax transformation

```
module Tiger-TraceAll
imports Tiger-Typed lib Tiger-Simplify
strategies
  instrument = topdown(try(TraceProcedure + TraceFunction));
               IntroducePrinters; simplify
rules
  TraceProcedure :
    [[ function f(xs) = e ]] ->
    [[ function f(xs) = (enterfun(s); e; exitfun(s)) ]]
    where !f => s
  TraceFunction :
    [[ function f(xs) : tid = e ]] ->
    [[ function f(xs) : tid =
         (enterfun(s);
          let var x : tid := nil in x := e; exitfun(s); x end) ]]
    where new => x ; !f => s
  IntroducePrinters :
    e -> [[ let var ind := 0
                function enterfun(name : string) = (
                  ind := +(ind, 1);
                  for i := 2 to ind do print(" ");
                  print(name); print(" entry\\n"))
                function exitfun(name : string) = (
                  for i := 2 to ind do print(" ");
                  ind := -(ind, 1);
                  print(name); print(" exit\\n"))
            in e end ]]
```

Concrete syntax transformation

Implementing Concrete Object Syntax

```
module StrategoTiger
imports
  Tiger Tiger-Sugar Tiger-Variables Tiger-Congruences
imports
  Stratego [ Id => StrategoId
         Var => StrategoVar
             StrChar => StrategoStrChar ]
exports
  context-free syntax
    "[[" Dec       "]]"  -> Term        {cons("ToTerm"),prefer}
    "[[" FunDec    "]]"  -> Term        {cons("ToTerm"),prefer}
    "[[" Exp       "]]"  -> Term        {cons("ToTerm"),prefer}
    "~"  Term            -> Exp         {cons("FromTerm"),prefer}
    "~*" Term            -> {Exp ","}+  {cons("FromTerm")}
    "~*" Term            -> {Exp ";"}+  {cons("FromTerm")}
    "~"  Term            -> Id          {cons("FromTerm")}
    "~*" Term            -> {FArg ","}+ {cons("FromTerm")}
```

Embedding of object language into meta language

From Concrete Syntax to Abstract Syntax

```
[[ x := let ds in ~* es end ]] -> [[ let ds in x := (~* es) end ]]
```

parse

```
Rule(ToTerm(Assign(Var(meta-var("x")),
                   Let(meta-var("ds"),FromTerm(Var("es"))))),
     ToTerm(Let(meta-var("ds"),
                [Assign(Var(meta-var("x")),
                        Seq(FromTerm(Var("es"))))])))
```

Mixed AST

explode

```
Rule(Op("Assign",[Op("Var",[Var("x")]),
                  Op("Let",[Var("ds"),Var("es")])]),
     Op("Let",[Var("ds"),
               Op("Cons",[Op("Assign",[Op("Var",[Var("x")]),
                                       Op("Seq",[Var("es")])]),
                          Op("Nil",[])])]))
```

Pure AST

pretty-print

```
Assign(Var(x),Let(ds,es)) -> Let(ds,[Assign(Var(x),Seq(es))])
```

Meta Explode

```
module meta-explode
imports lib Stratego
strategies
  meta-explode =
    alltd(?ToTerm(<trm-explode>) + ?ToStrategy(<str-explode>))

  trm-explode =
    TrmMetaVar <+ TrmStr <+ TrmFromTerm <+ TrmFromStr <+ TrmAnno
    <+ TrmConc <+ TrmNil <+ TrmCons <+ TrmOp

  TrmOp       : op#(ts) -> Op(op, <map(trm-explode)> ts)

  TrmMetaVar  : meta-var(x) -> Var(x)
  TrmStr      = is-string; !Str(<id>)
  TrmFromTerm = ?FromTerm(<meta-explode>)
  TrmFromStr  = ?FromStrategy(<meta-explode>)
  TrmAnno     = Anno(trm-explode, meta-explode)
  TrmNil      : [] -> Op("Nil", [])
  TrmCons     : [x | xs] -> Op("Cons",[<trm-explode>x, <trm-explode>xs])
  TrmConc     : Conc(ts1,ts2) ->
                <foldr(!<trm-explode> ts2,
                       !Op("Cons", [<Fst>, <Snd>]), trm-explode)> ts1
```

Find term embedding

Explode it

How do you type check that?

The concrete syntax embedding techniques is not specific to Stratego as meta-language. This paper shows how to use it to embed DSLs into Java.

```
ATerm x = id |[ propertyChangeListeners ]|;

ATerm stm = bstm |[ {
    if(x == null) return;
    PropertyChangeEvent event =
      new PropertyChangeEvent(this, f, v1, v1);
    for(int c=0; c < x.size(); c++) {
      ((...)x.elementAt(c)).propertyChange(event);
    }
  }
]|;
```

https://doi.org/10.1145/1035292.1029007

Concrete Syntax for Objects

Domain-Specific Language Embedding and Assimilation without Restrictions

Martin Bravenboer
Institute of Information and Computing Sciences
Universiteit Utrecht, P.O. Box 80089
3508 TB Utrecht, The Netherlands
martin@cs.uu.nl

Eelco Visser
Institute of Information and Computing Sciences
Universiteit Utrecht, P.O. Box 80089
3508 TB Utrecht, The Netherlands
visser@acm.org

ABSTRACT

Application programmer's interfaces give access to domain knowledge encapsulated in class libraries without providing the appropriate notation for expressing domain composition. Since object-oriented languages are designed for extensibility and reuse, the language constructs are often sufficient for expressing domain abstractions at the semantic level. However, they do not provide the right abstractions at the syntactic level. In this paper we describe MetaBorg, a method for providing *concrete syntax* for domain abstractions to application programmers. The method consists of *embedding* domain-specific languages in a general purpose host language and *assimilating* the embedded domain code into the surrounding host code. Instead of extending the implementation of the host language, the assimilation phase implements domain abstractions in terms of existing APIs leaving the host language undisturbed. Indeed, MetaBorg can be considered a method for promoting APIs to the language level. The method is supported by proven and available technology, i.e. the syntax definition formalism SDF and the program transformation language and toolset Stratego/XT. We illustrate the method with applications in three domains: code generation, XML generation, and user-interface construction.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.3 [**Programming Languages**]: Processors

General Terms: Languages, Design

Keywords: MetaBorg, Stratego, SDF, Embedded Languages, Syntax Extension, Extensible Syntax, Domain-Specific Languages, Rewriting, Meta Programming, Concrete Object Syntax

1. INTRODUCTION

Class libraries encapsulate knowledge about the domain for which the library is written. The application programmer's interface to a library is the means for programmers to access that knowledge. However, the generic language of method invocation provided by object-oriented languages does often not provide the right notation for expressing domain-specific composition. General purpose languages, particularly object-oriented languages, are designed for extensibility and reuse. That is, language concepts such as objects, interfaces, inheritance, and polymorphism support the construction of class hierarchies with reusable implementations that can easily be extended with variants. Thus, OO languages provide the flexibility to develop and evolve APIs according to growing insight into a domain.

Although these facilities are often sufficient for expressing domain abstractions at the semantic level, they do not provide the right abstractions at the syntactic level. This is obvious when considering the domain of arithmetic or logical operations. Most modern languages provide infix operators using the well known notation from mathematics. Programmers complain when they have to program in a language where arithmetic operations are made available in the same syntax as other procedures. Consider writing `e1 + e2` as `add(e1, e2)` or even `x := e1; x.add(e2)`. However, when programming in other domains such as code generation, document processing, or graphical user-interface construction, programmers are forced to express their designs using the generic notation of method invocation rather than a more appropriate domain notation. Thus programmers have to write code such as

```
JPanel panel =
  new JPanel(new BorderLayout(12,12));
panel.setBorder(
  BorderFactory.createEmptyBorder(15,15,15,15));
```

in order to construct a user-interface, rather than using a more compositional syntax reflecting the nice hierarchical structure of user-interface components in the Swing library. Building in syntactic support for such domains in a general purpose language is not feasible, however, because of the different speeds at which languages and domain abstractions develop. A language should strive for stability, while libraries can be more volatile.

In this paper we describe MetaBorg, a method for providing *concrete syntax* for domain abstractions to application programmers. The method consists of *embedding*

This paper generalizes the concrete syntax techniques to all sorts of host and guest languages, with an application to preventing injection attacks.

Injection attacks are caused by unhygienic construction of code through which user input can be turned into executable code.

doi:10.1016/j.scico.2009.05.004

Science of Computer Programming 75 (2010) 473–495

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

ELSEVIER

Preventing injection attacks with syntax embeddings☆

Martin Bravenboer [a,*], Eelco Dolstra [b], Eelco Visser [b]

[a] Department of Computer Science, University of Massachusetts Amherst, 140 Governors Drive, Amherst, MA 01003, USA

[b] Department of Software Technology, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands

A R T I C L E I N F O

Article history:
Received 7 March 2008
Received in revised form 18 May 2009
Accepted 21 May 2009
Available online 31 May 2009

Keywords:
Injection attacks
Security
Syntax embedding
Program generation
Program transformation
Concrete object syntax

A B S T R A C T

Software written in one language often needs to construct sentences in another language, such as SQL queries, XML output, or shell command invocations. This is almost always done using *unhygienic string manipulation*, the concatenation of constants and client-supplied strings. A client can then supply specially crafted input that causes the constructed sentence to be interpreted in an unintended way, leading to an *injection attack*. We describe a more natural style of programming that yields code that is impervious to injections *by construction*. Our approach embeds the grammars of the *guest languages* (e.g. SQL) into that of the *host language* (e.g. Java) and automatically generates code that maps the embedded language to constructs in the host language that reconstruct the embedded sentences, adding escaping functions where appropriate. This approach is generic, meaning that it can be applied with relative ease to any combination of context-free host and guest languages.

1. Introduction

In this paper we propose using *syntax embedding* to prevent injection vulnerabilities in a language-independent way. Injections form a very common class of security vulnerabilities [22]. Software written in one language often needs to construct sentences in another language, such as SQL, XQuery, or XPath queries, XML output, or shell command invocations. This is almost always done using *unhygienic string manipulation*, whereby constant and client-supplied strings are concatenated to form the sentence. Consider for example the following piece of server-side Java code that authenticates a remote HTTP user against a database, where getParam() returns a string supplied by the user, for instance through a form field:

```
String userName = getParam("userName");
String password = getParam("password");
String query = "SELECT id FROM users "
             + "WHERE name = '" + userName + "' "
             + "AND password = '" + password + "'";
if (executeQuery(query).size() == 0)
  throw new Exception("bad user/password");
```

On testing, this code may appear to work correctly, but it is vulnerable to a very common security flaw. For instance, if the user specifies as the password the string `' OR 'x' = 'x`, then the constructed SQL query will be

```
SELECT id FROM users WHERE name = '...' AND password = '' OR 'x' = 'x'
```

☆ An earlier version appeared in GPCE '07: Proceedings of the 6th International Conference on Generative Programming and Component Engineering.

* Corresponding author.
E-mail addresses: martin.bravenboer@acm.org (M. Bravenboer), e.dolstra@tudelft.nl (E. Dolstra), visser@acm.org (E. Visser).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.05.004

Hygienic

```
$username = $_GET['username'];
$q = "SELECT * FROM users WHERE username = '" . $username . "'";
executeSQL($q);
```

SQL in PHP: SQL injection vulnerability

```
String e = "/users[@name='" + name + "' and " +
           "@password='" + password + "']";
factory.newXPath().evaluate(e, doc);
```

XPath in Java: XPath injection vulnerability

```
$searchfilter = "(cn=" . $username . ")";
$search = ldap_search($connection, $directory, $searchfilter);
```

LDAP in PHP: LDAP injection vulnerability

```
$command = "svn cat \"file name\" -r" . $rev;
system($command);
```

Shell calls in PHP: command injection vulnerability

```
String topic = getParam("topic");
String query = "SELECT body FROM comments WHERE topic = '" + topic + "'";
ResultSet results = executeQuery(query);
foreach (String body : results)
  println("<tr><td>" + body + "</td></tr>");
```

XML and SQL in Java: XSS vulnerability

```
$username = $_GET['username'];
$q = <| SELECT * FROM users WHERE username = ${$username} |>;
executeSQL($q->toString());
```

SQL in PHP

```
XPath e = {- /users[@name=${name} and @password=${password}] -};
factory.newXPath().evaluate(e.toString(), doc);
```

XPath in Java

```
$searchfilter = (| (cn=$($username)) |);
$search = ldap_search($connection, $directory, $searchfilter->toString());
```

LDAP in PHP

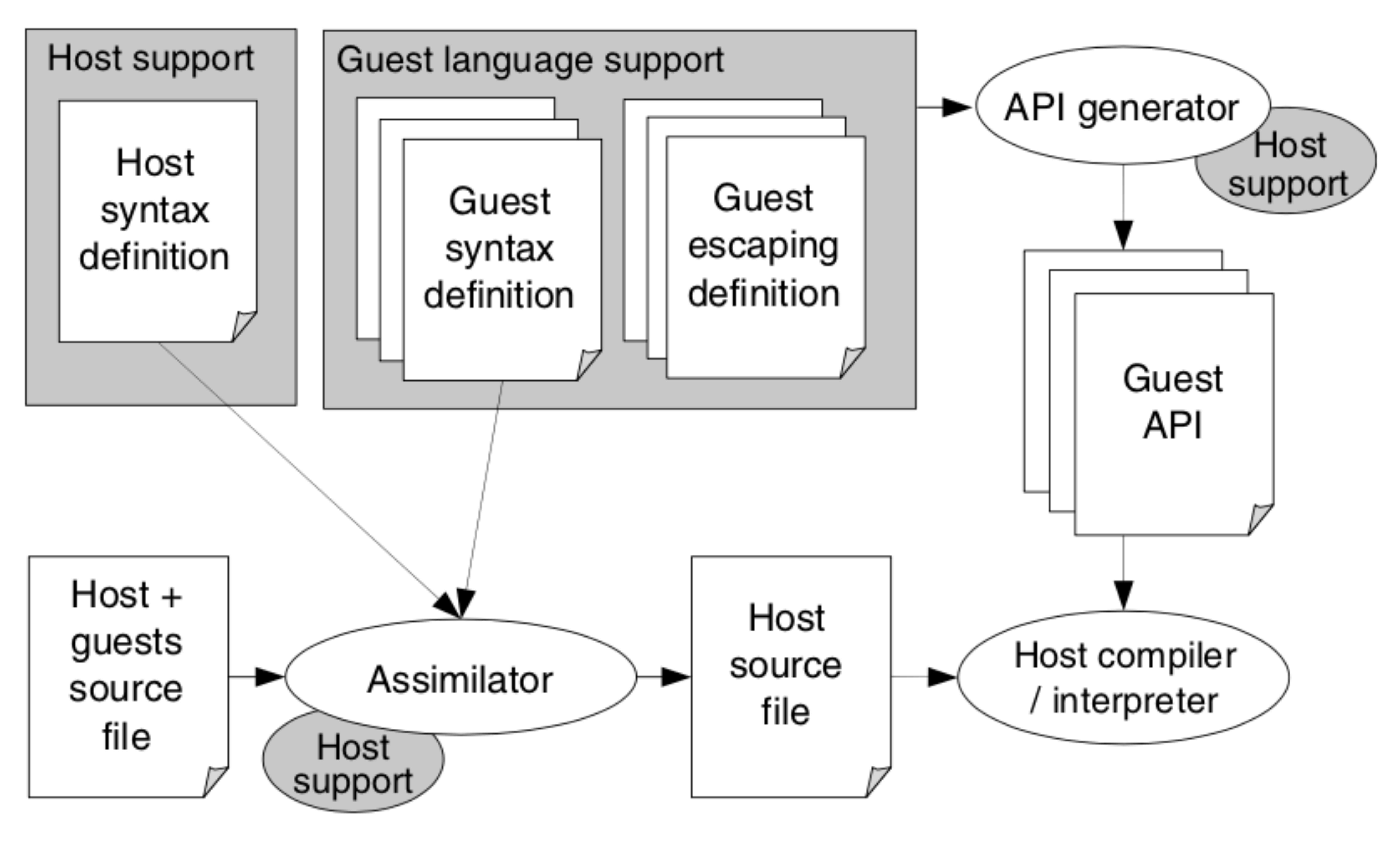
```
$command = <| svn cat "file name" -r${$rev} |>;
system($command->toString());
```

Shell calls in PHP

```
String topic = getParam("topic");
SQL query = <| SELECT body FROM comments WHERE topic = ${topic} |>;
ResultSet results = executeQuery(query.toString());
foreach (String body : results)
  println(<tr><td>${body}</td></tr>.toString());
```

XML and SQL in Java

A Generic Architecture



Hygienic Transformations

Hygienic Transformations

```
module Tiger-TraceAll
imports Tiger-Typed lib Tiger-Simplify
strategies
  instrument = topdown(try(TraceProcedure + TraceFunction));
               IntroducePrinters; simplify
rules
  TraceProcedure :
    FunDec(f, xs, NoTp, e) ->
    FunDec(f, xs, NoTp,
           Seq([Call(Var("enterfun"),[String(f)]), e,
                Call(Var("exitfun"),[String(f)])]))
  TraceFunction :
    FunDec(f, xs, Tp(tid), e) ->
    FunDec(f, xs, Tp(tid),
           Seq([Call(Var("enterfun"),[String(f)]),
                Let([VarDec(x,Tp(tid),NilExp)],
                    [Assign(Var(x), e),
                     Call(Var("exitfun"),[String(f)]),
                     Var(x)])]))
    where new => x
  IntroducePrinters :
    e -> /* omitted for brevity */
```

Does new variable in TraceProcedure not capture variables in e?

Guaranteeing Hygiene

Guarantee that variables are not captured

- Which variables?

Object language name analysis for transformation rules

- E.g. apply Tiger constraint rules to patterns in rules

Existing approaches

- Hygienic macros in Scheme/Racket
- Higher-order abstract syntax
- Nominal abstract syntax

Research

- Hygienic transformations for more complex binding patterns

Guaranteeing Type Correct Target Code

Guaranteeing Type Correct Code

Property: Type correct target code

- Guarantee that generated code type checks

Intrinsically-typed ASTs

- Encode type system in algebraic signature
- Including binding structure
- Language support: Generalized ADTs

Research

- Advanced type systems & binding patterns

Semantics Preservation

Interface Preservation

Generate code has same interface as source code

Type Preservation

Generated code produces values with the same type

Intrinsically-typed interpreters for imperative languages

- POPL18 paper
- Verify that interpreters are type preserving
- Including non-lexical binding patterns

Research

- how to do this for other transformations?

Dynamic Semantics Preservation

Semantics preservation

- Generated code has the same behaviour as the source program

CompCert

- Certified C compiler
- Defines operational semantics of source language (most of C) and all intermediate languages
- Mechanically verify that translations between IR preserve behaviour
  - For all possible programs
- Or: verify that generated output has same behaviour as input
  - For programs that compiler is applied to