Secure by Construction

Lecture 7

CS4105 - Software Security

Q2 / 2015 – 2016

Eelco Visser

TU Delft

Buffer Overflow Vulnerability

```
void main(int argc, char **argv) {
  char program_name[256];
  strcpy(program_name, argv[0]);
  f(program_name);
}
```

```
void strcpy(char *dst, char *src) {
  int i = 0;
  do {
    dst[i] = src[i];
  } while (src[i++] != '\0')
}
```

Problem: argv[0] may not fit in program_name

Violates: memory safety / stack safety

Buffer API: Size Tracking Built-In

```
typedef struct {
  char* ptr;
  int bufsize;
} buffer;
buffer *alloc_buf(int size) {
  buffer *buf = (buffer *)malloc(sizeof(buffer));
  buf->bufsize = size;
  buf->ptr = (char *)malloc(buf->bufsize);
  return buf;
}
buffer *copy(buffer *src) {
  buffer *dst = alloc_buf(src->bufsize);
  strncpy(dst->ptr, src->ptr, dst->bufsize);
  dst->ptr[dst->bufsize-1] = '\0';
  return dst;
}
```

Invariant: bufsize is the size of the buffer assigned to ptr

API maintains invariant

Problem: C does not enforce API

`buff.ptr[buf.bufsize]`

Secure by Construction

Secure by Construction

- Verify that invariants of abstraction are maintained
- Only generate code that is guaranteed to be safe
- No unsafe escapes

This Lecture and Assignment I3

- Study how this works
- Toy language: BufferC
- Safe ‘buffer’ abstraction
- Combination of static type checks and dynamic run-time library checks guarantee safety
- Implementation in Spoofax Language Workbench

Assignment I3: Secure by Construction

Complete basic implementation of BufferC

- **Code generation** rules for buffer operations
- Adapt code generation to realize **reference counting garbage collection**
- Implement a **run-time** library buffer.c/buffer.h that provides an implementation of the primitive operations for buffers
- A library `**libstring.bc**` in BufferC that implements a number of standard string operations
- Write **test programs** in BufferC to test the correct behavior (absence of vulnerabilities!) of your implementation

TU Delft

The BufferC Language

Subset of C language

- C functions definitions and calls
- Expressions
  - arithmetic, comparisons
  - *booleans* (`true`,`false`)
  - basic assignments (`e = e;`)
- Not included: arrays, pointers, structs, …

Modules

- module foo imports libstring { … }

Buffer data type

- Types: buffer, FILE
- String literals
- Operations: create, length, print, printf, fopen, fread
- Array access notation: `b[i]`

Expressions

```
int i;

int j = i + 1;

i < j;

j == i + 1;

j == 1 || j == 2;

!(true) || false;

j = i++;

char c = 'a';

c = '\n';

// etc.
```

TU Delft

Statements

```
if(!(i < j) || j == i)
  i = i++;
else
  i = 0;

while(i < j) i++;

do i++; while(i < j)

for(int k = 0; k < j; k++) i++;

for(i = 0; i < j; i++) i++;

return i + j;

exit(2);

{ i = i + 1; j = j - 1; }
```

TU Delft

Functions

```
int exp(int x, int n) {
  if(n == 0) {
    return 1;
  } else {
    return x * exp(x, n - 1);
  }
}
```

TU Delft

Buffer Operations

```
buffer a;                  // built-in buffer type

buffer b = create(5); // create empty buffer of length 5

b = "abcdefgh";            // buffer literal

b[4];                      // indexed access

b[4] = 'a';                // indexed assignment

print(b);                  // print content of buffer

printf("b: %s\n", b); // print with format string

b.length;                  // length of string

FILE f = fopen("a", "r"); // open file

fread(b, f);               // read file into buffer
```

TU Delft

Assignment: String Library

Implement a set of standard string functions in BufferC

```
buffer str_copy(buffer dst, buffer src) {
  for(int i = 0; i < dst.length && i < src.length; i++) {
    dst[i] = src[i];
  }
  return dst;
}
```

TU Delft

libstring.bc Interface

```
module libstring_interface {

  // str_copy(dst, src): copy characters of src buffer into dst buffer
  // and return dst
  buffer str_copy(buffer dst, buffer src);

  // str_clone(src) : create a new buffer that is an exact copy of src
  buffer str_clone(buffer src);

  // str_concat(s1, s2): concatenate buffers s1 and s2 into a new buffer
  buffer str_concat(buffer s1, buffer s2);

  // str_trim(s): remove leading and trailing spaces from s
  // i.e. copy s into a new buffer without leading and trailing spaces
  buffer str_trim(buffer s);

  // str_equal(s1, s2): true if s1 and s2 are exactly equal
  boolean str_equal(buffer s1, buffer s2);

  // str_compare(s1, s2): compare two strings lexicographically
  int str_compare(buffer s1, buffer s2);

  // str_error(msg): print an error message
  void str_error(buffer msg);
}
```

Assignment: Test Programs

Write BufferC programs to test the compiler

Good behavior

- Do generated programs behave as expected?

Bad behavior

- Do potentially bad programs lead to triggering dynamic checks?
- Out of bound buffer access
- Safe garbage collection
- Format string vulnerabilities

TU Delft

A Complete Example

```
module vulnerable imports libstring {

   int bof(buffer str) {
      buffer dst = create(24);
      str_copy(dst, str);
      return 1;
   }

   int main(buffer filename) {
      buffer str = create(517);
      FILE badfile = fopen(filename, "r");
      fread(str, badfile);
      bof(str);
      print("Returned Properly\n");
      exit(0);
   }

}
```

Language Definition in Spoofax

Meta-languages

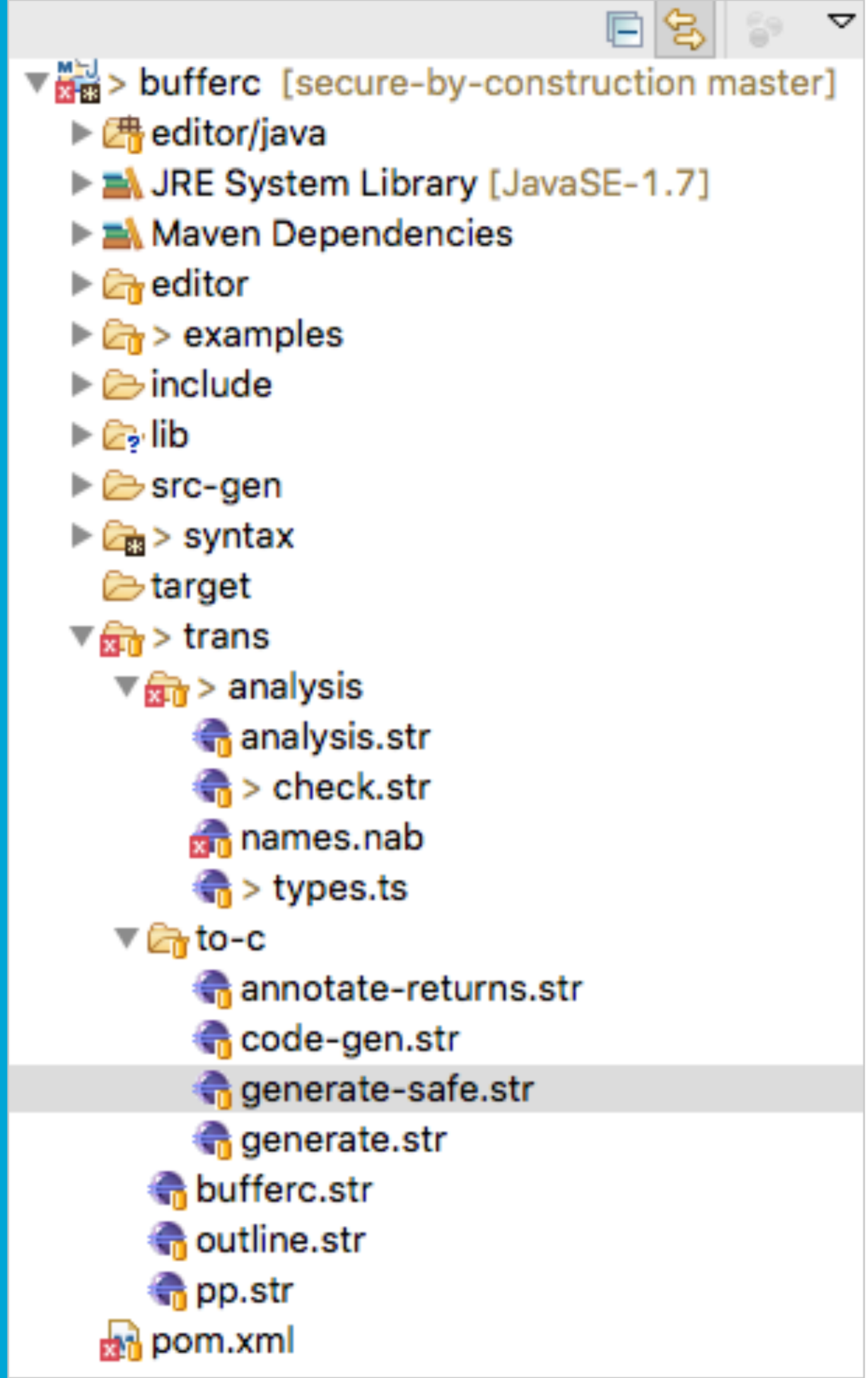
- Syntax definition
- Name binding rules
- Type rules
- Code generation rules

Target language

- Run-time system

Test programs

TU Delft


> bufferc [secure-by-construction master]
editor/java
JRE System Library [JavaSE-1.7]
Maven Dependencies
editor
> examples
include
lib
src-gen
> syntax
target
> trans
> analysis
analysis.str
> check.str
names.nab
> types.ts
to-c
annotate-returns.str
code-gen.str
generate-safe.str
generate.str
bufferc.str
outline.str
pp.str
pom.xml

TU Delft

Syntax Definition

Defining well-formed programs

- Context-free grammars
- Disambiguation
- Abstract syntax trees (terms)

TU Delft

Context-Free Grammar Productions

```
context-free syntax

  Exp.Multiplication    = <<Exp> * <Exp>> {left}
  Exp.Division          = <<Exp> / <Exp>> {left}
  Exp.Modulo            = <<Exp> % <Exp>> {left}
  Exp.Addition          = <<Exp> + <Exp>> {left}
  Exp.Subtraction       = <<Exp> - <Exp>> {left}
```

Disambiguation

```
context-free syntax

  Exp.Multiplication    = <<Exp> * <Exp>> {left}
  Exp.Division          = <<Exp> / <Exp>> {left}
  Exp.Modulo            = <<Exp> % <Exp>> {left}
  Exp.Addition          = <<Exp> + <Exp>> {left}
  Exp.Subtraction       = <<Exp> - <Exp>> {left}
```

```
context-free priorities

  {left :
    Exp.Multiplication
    Exp.Division
    Exp.Modulo
  } > {left :
    Exp.Addition
    Exp.Subtraction
  }
```

Abstract Syntax Trees (Terms)

```
if(j == 1 || j == 2 + i) {
  print("ok");
}
```

parse

```
If(
  Or(
    Equal(Var(Identifier("j")), Decimal("1"))
    , Equal(
        Var(Identifier("j"))
      , Addition(Decimal("2"), Var(Identifier("i")))
      )
    )
    , Block([Exp(FunctionCall(Print(), [String("\"ok\"")]))])
  )
```

Abstract Syntax Trees (Terms)

```
FunDef(
  Int()
  , Identifier("exp")
  , [ Param(Int(), Decl(Identifier("x")))
    , Param(Int(), Decl(Identifier("n")))
    ]
  , [ IfElse(
        Equal(Var(Identifier("n")), Decimal("0"))
      , Block([Return(Some(Decimal("1")))])
      , Block(
          [ Return(
              Some(
                Multiplication(
                  Var(Identifier("x"))
                , FunctionCall(
                    Identifier("exp")
                  , [Var(Identifier("x")), Subtraction(Var(Identifier("n")), Decimal("1"))]
                  )
                )
              )
            )
          ]
        )
      )
    ]
  )
```

```
int exp(int x, int n) {
  if(n == 0) {
    return 1;
  } else {
    return x * exp(x, n - 1);
  }
}
```

Java - bufferc/examples/abstractsyntax.bc - Eclipse - /Users/eelcovisser/Dropbox/03-Research/workspace-spx-apps

Quick Access | Java

Expressions.sdf3

```
module Expressions

imports Identifiers
imports Constants
imports Types

context-free syntax

  Exp.Var               = <<Identifier>>
  Exp                   = <<Constant>>
  Exp.CommaExp          = <(<CommaExp>)>
  Exp.FunctionCall      = <<Identifier>(<{Exp ", "}*>)>
  Exp.ArrayField        = <<Exp>[<IndexExp>]>
  Exp.Field             = <<Exp>.<Identifier>>
  Exp.IncrementPostfix  = <<Exp>++> {left}
  Exp.DecrementPostfix  = <<Exp>--> {left}

  IndexExp = Exp

  Exp.Not = <!<Exp>> {right}

context-free syntax

  Exp.Multiplication    = <<Exp> * <Exp>> {left}
  Exp.Division          = <<Exp> / <Exp>> {left}
  Exp.Modulo            = <<Exp> % <Exp>> {left}
  Exp.Addition          = <<Exp> + <Exp>> {left}
  Exp.Subtraction       = <<Exp> - <Exp>> {left}

  Exp.LessThan          = <<Exp> \<   <Exp>> {left}
  Exp.LessThanEqual     = <<Exp> \<=  <Exp>> {left}
  Exp.GreaterThan       = <<Exp> \>   <Exp>> {left}
  Exp.GreaterThanEqual  = <<Exp> \>=  <Exp>> {left}
```

abstractsyntax.bc

```
module abstractsyntax {
  void abstract_syntax(int i, int j) {
    if(j == 1 || j == 2 + i) {
      print("ok");
    }
  }
}
```

abstractsyntax.aterm

```
Module(
  Identifier("abstractsyntax")
, None()
, [ FunDef(
      Void()
    , Identifier("abstract_syntax")
    , [ Param(Int(), Decl(Identifier("i")))
      , Param(Int(), Decl(Identifier("j")))
      ]
    , [ If(
          Or(
            Equal(Var(Identifier("j")), Decimal("1"))
          , Equal(
              Var(Identifier("j"))
            , Addition(Decimal("2"), Var(Identifier("i")))
            )
          )
        , Block([Exp(FunctionCall(Print(), [String("\"ok\"")]))])
        )
      ]
    )
  ]
)
```

Writable | Insert | 5 : 6

The Buffer Extension

```
module Buffers

imports Types
imports Identifiers
imports Keywords

context-free syntax

  Type.Int    = <int>
  Type.Buffer = <buffer>
  Type.FILE   = <FILE>

  Identifier.Length = <length>
  Identifier.Create = <create>
  Identifier.Size   = <size>
  Identifier.Print  = <print>
  Identifier.PrintF = <printf>
  Identifier.Fopen  = <fopen>
  Identifier.Fread  = <fread>
```

```
[ DeclStm(Decl(Buffer(), Decl(Identifier("a"))))
, DeclStm(
    DeclInit(
      Buffer()
    , Decl(Identifier("b"))
    , FunctionCall(Create(), [Decimal("5")])
    )
  )
, Exp(Assign(Var(Identifier("b")), Assign(), String("\"abcdefgh\"")))
, Exp(ArrayField(Var(Identifier("b")), Decimal("4")))
, Exp(
    Assign(
      ArrayField(Var(Identifier("b")), Decimal("4"))
    , Assign()
    , Char("'a'")
    )
  )
, Exp(FunctionCall(Print(), [Var(Identifier("b"))]))
, Exp(
    FunctionCall(PrintF(), [String("\"b: %s\\n\""), Var(Identifier("b"))])
 )
, Exp(Field(Var(Identifier("b")), Length()))
, DeclStm(
    DeclInit(
      FILE()
    , Decl(Identifier("f"))
    , FunctionCall(Fopen(), [String("\"a\""), String("\"r\"")])
    )
  )
, Exp(
    FunctionCall(
      Fread()
    , [Var(Identifier("b")), Var(Identifier("f"))]
    )
  )
]
```

```
buffer a;

buffer b = create(5);

b = "abcdefgh";

b[4];

b[4] = 'a';

print(b);

printf("b: %s\n", b);

b.length;

FILE f = fopen("a", "r");

fread(b, f);
```

Name Binding Rules

Name consistency

- Check that declarations are unique (in a scope)
- Check that references are in scope of a declaration
- Also: navigation (and completion)

Name binding concepts

- defines: declaration of a name
- refers to: named reference to a declaration
- scopes: limit the visibility of declarations
- imports: make declarations from another scope visible

Associated types

- of type: associate type with a declaration
- has type: retrieve type of an expression

Name Binding Rules: Variables

```
binding rules // variables

  Param(t, Decl(Identifier(name))) :
    defines Variable name of type ty
    where t has type ty

  Decl(t, Decl(Identifier(name))) :
    defines Variable name of type ty
    where t has type ty

  DeclInit(t, Decl(Identifier(name)), e) :
    defines Variable name of type ty
    where t has type ty

  Var(Identifier(name)) :
    refers to Variable name

  ForDec(_, _, _, _) :
    scopes Variable
```

Name Binding Rules: Functions

```
binding rules // functions

  FunProto(t, Identifier(name), params) :
    defines Function name of type (ty, tys_param)
    where t has type ty
      and params has type tys_param
    scopes Variable

  FunDef(t, Identifier(name), params, stms) :
    defines Function name of type (ty, tys_param)
    where t has type ty
      and params has type tys_param
    scopes Variable

  FunctionCall(Identifier(name), es) :
    refers to Function name
```

Name Binding Rules: Modules

```
binding rules // modules

  Module(Identifier(name), _, _) :
    defines Module name
    scopes Variable, Function

  Import(Identifier(name)) :
    imports Function from Module name
```

Type Rules

Type consistency

- Check that arguments of operands are compatible with operators
- Check that function arguments are compatible with function definition

Interaction with name binding

- definition of: refer to (a property of) the declaration of a reference
- type-dependent name resolution
  - type of field access to struct
  - (not needed for BufferC)

TU Delft

Syntax of Type Rules

```
type rules

  C(e1, e2) : t
    where e1: t
      and t == Int()
          else error "message" on e1
```

Type Rules: Operators

```
type rules // operators

  Addition(e1, e2) : Int()
    where e1: ty1
      and e2: ty2
      and ty1 == Int()
          else error "int expected" on e1
      and ty2 == Int()
          else error "int expected" on e2
```

Type Rules: Types and Constants

```
type rules // types

   Void()   : Void()
   Int()    : Int()
   Buffer() : Buffer()
   Bool()   : Bool()
   Char()   : Char()
   FILE()   : FILE()

type rules // constants

   Decimal(val) : Int()
   True()       : Bool()
   False()      : Bool()
   Char(val)    : Char()
   String(val)  : Buffer()
   Null()       : Buffer()
```

Type Rules: Variables

```
type rules // variables

  Param(t, Decl(Identifier(name))): ty
  where t : ty

  Var(Identifier(x)) : t
  where definition of x : t

  DeclInit(t, Decl(Identifier(x)), e) :-
  where e : e_ty
    and t : ty
    and e_ty == ty
        else error $[[ty] expected] on e
```

Type Rules: Function Calls

```
type rules // function calls

   e@FunctionCall(Identifier(name), es): ty
   where definition of name: (ty, tys)
     and es : tys_es
     and tys == tys_es
         else error "argument types not compatible" on e
```

Type Rules: Buffer Operations (1)

```
type rules // buffer operations

   FunctionCall(Create(), [e]): Buffer()
   where e : Int()
         else error "int expected" on e

   FunctionCall(Print(), [e]) : Void()
   where e : Buffer()
         else error "buffer expected" on e

   FunctionCall(PrintF(), [e1, e2]) : Void()

   FunctionCall(Fopen(), [e1, e2]) : FILE()
   where e1 : Buffer()
         else error "buffer expected" on e1
    and e2 : Buffer()
         else error "buffer expected" on e2

   FunctionCall(Fread(), [e1, e2]) : Void()
   where e1 : Buffer()
         else error "buffer expected" on e1
    and e2 : FILE()
         else error "buffer expected" on e2
```

Type Rules: Buffer Operations (2)

```
type rules // array access

  Field(e, Length()): Int()
  where e : Buffer()
        else error "buffer expected" on e

  ArrayField(e1, e2) : Char()
  where e1 : Buffer()
       else error "buffer expected" on e1
   and e2 : Int()
       else error "int expected" on e2

Assign(ArrayField(e1, e2), Assign(), e3) : Char()
where e1 : Buffer()
      else error "buffer expected" on e1
 and e2 : Int()
      else error "int expected" on e2
 and e3 : Char()
      else error "char expected" on e3

Assign(Var(Identifier(x)), Assign(), e) : ty
where definition of x : ty
  and e : ty_e
  and ty_e == ty
       else error $[[ty] expected] on e
```

Dynamic Semantics

Dynamic semantics of BufferC defined by translation to C

Code generator

- Translates BufferC abstract syntax tree of module m.bc to
- C compilation unit m.c
- C header file m.h

Run-time system

- Implementation of primitive operations in C

Type Soundness

Requirement: BufferC is type sound

A BufferC program p that passes the type checker:

- Does not allow out of bound memory access
- Does not free live buffers (no dangling references)
- Does free dead buffers (no memory leaks)

Not all properties can be guaranteed statically

- Acceptable: end with error (exception)
- Not acceptable: unsafe memory access

Code Generation

Code generator

- Translates high-level language abstractions to low-level target language
- Use abstractions from buffer.c (instead of generating low-level code)
- Generates run-time checks
- Generates implicit operations (e.g. reference counting)

Concepts

- Term rewrite rules
- Pattern matching
- String templates

Signature: Abstract Syntax Constructors

```
signature
  constructors
    False : Constant
    True  : Constant
          : Constant -> Exp
    Not   : Exp -> Exp
    Or    : Exp * Exp -> Exp
    And   : Exp * Exp -> Exp
```

Term Rewrite Rules

```
rules

  eval-and :
    And(False(), e) -> False()

  eval-and :
    And(True(), e) -> e

  eval-or :
    Or(False(), e) -> e

  eval-or :
    Or(True(), e) -> True()

  eval-not :
    Not(True()) -> False()

  eval-not :
    Not(False()) -> True()

  double-negation :
    Not(Not(e)) -> e

  de-morgan :
    Not(And(e1, e2)) -> Or(Not(e1), Not(e2))
```

Rewriting Strategies

```
strategies

  disjunctive-normal-form =
    innermost(
      eval-and
    <+ eval-or
    <+ eval-not
    <+ double-negation
    <+ de-morgan
    )
```

Recursive Rewrite Rules

```
rules

  trans :
    True() -> Decimal("1")
  trans :
    False() -> Decimal("0")
  trans :
    And(e1, e2) -> Multiplication(<trans>e1, <trans>e2)
  trans :
    Or(e1, e2) -> Addition(<trans>e1, <trans>e2)
  trans :
    Not(e) -> Subtraction(Modulo(<trans>e, Decimal("1")), Decimal("1"))
```

Builders for Code Generation

```
module code-gen

imports
   to-c/generate
   to-c/generate-safe

rules // builders for code generation

   generate-c :
      (selected, position, ast, path, project-path) -> (filename, result)
      with
         filename := <guarantee-extension(|"c")> path;
         result    := <generate-safec>ast

   generate-h :
      (selected, position, ast, path, project-path) -> (filename, result)
      with
         filename := <guarantee-extension(|"h")> path;
         result    := <generate-safec-header>ast
```

Generating C

```
module generate

imports include/BufferC

rules

   genc =
      gen-c-special
      <+ gen-c
      <+ ugly-print

   genc-header =
      gen-c-header-special
      <+ gen-c-header
      <+ ugly-print

   gen-c-special =
      fail

   gen-c-header-special =
      fail
```

```
rules

   ugly-print :
      c#(ts) ->
      $[
         [c](
            [<mapsep(genc|", ")>ts]
         )]

   mapsep(f | sep) =
      map(f); separate-by(|sep)
```

Generating Strings

```
rules // types

  gen-c : Void()    -> "void"
  gen-c : Int()     -> "int"
  gen-c : Char()    -> "char"
  gen-c : Bool()    -> "int"
  gen-c : FILE()    -> "FILE *"
```

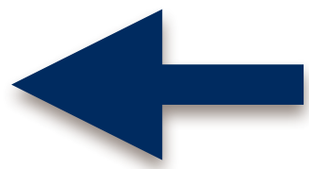
```
  gen-c :
   If(e, s) -> <concat-strings>[
     "if(", <genc>e, ")\n",
        <genc>s, "\n"
   ]
```

convenient, but …

… quoting, escaping, layout

String Templates

```
gen-c :
If(e, s) -> $[
  if([<genc>e])
    [<genc>s]
]
```



```
gen-c :
If(e, s) -> <concat-strings>[
  "if(", <genc>e, ")\n",
    <genc>s, "\n"
]
```

```
$[ ...
   ...
   ... ]
```

quote literal multi-line text

```
$[ ... [<genc>e] ... ]
```

escape to host language

Generation Scheme by Example

```
module base imports libstring {
  int exp(int x, int n) {
    if(n == 0) {
      return 1;
    } else {
      return x * exp(x, n - 1);
    }
  }
}
```

base.bc

```
#include <stdlib.h>
#include "buffer.h"

int exp(int x, int n);
```

base.h

```
#include <stdlib.h>
#include "buffer.h"
#include "base.h"
#include "libstring.h"

int exp(int x, int n) {
   if ((n == 0)) {
      return 1;
   } else {
      return (x * exp(x, (n - 1)));
   }
}
```

base.c

Generation for Base Language is (mostly) Homomorphic

```
gen-c :
  If(e, Block(stms)) -> $[
     if([<genc>e]) {
        [<gen-c-stms>stms]
     }
    ]


gen-c :
  IfElse(e, Block(s1), Block(s2)) -> $[
     if([<genc>e]) {
        [<gen-c-stms>s1]
     } else {
        [<gen-c-stms>s2]
     }]
```

Assignment: Run-Time Library and Code Generation

```
buffer x;                    // built-in buffer type

create(e);                   // buffer creation

"abcdefgh";                  // buffer literal

e1[e2];                      // indexed access

e1[e2] = e3;                 // indexed assignment

print(e);                    // print content of buffer

printf(e1, e2);              // print with format string

e.length;                    // length of string

FILE f = fopen(e1, e2); // open file

fread(e1, e2);               // read file into buffer
```

C abstraction (type, function) + generation rule for each

Code Generation for Buffer Operations

```
module generate-safe

imports include/BufferC
imports to-c/generate
imports to-c/annotate-returns
imports runtime/types/-
imports runtime/editor/annotations

rules // builders

   generate-safec = annotate-all-returns; genc

   generate-safec-header = genc-header

rules // buffer operations

   gen-c-special :
      Buffer() -> $[buffer *]

   gen-c-special :
      Null() -> $[NULL]

   gen-c-special :
     OtherConstruct(…) -> $[…]
```

Reference Counting

Goal

- Safe garbage collection
- Reliably release allocated buffers
- No dangling pointers
- No memory leaks

Technique

- Reference counting
- Count how many reference to buffer are live
- Increase counter when making a copy
- Decrease counter when deleting a copy
- Free buffer when counter reaches zero

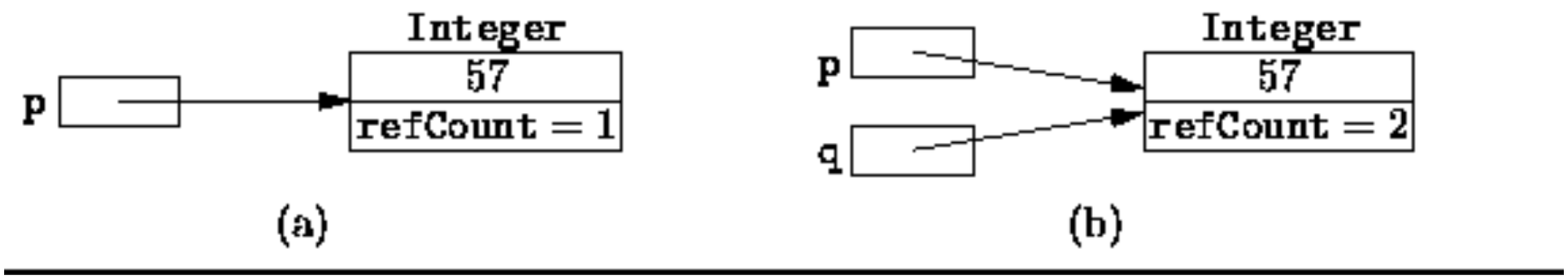
Instrumentation

- Inject increment and decrement operations in generated code

Source: http://www.brpreiss.com/books/opus5/html/page421.html

Increment on Copy of Reference

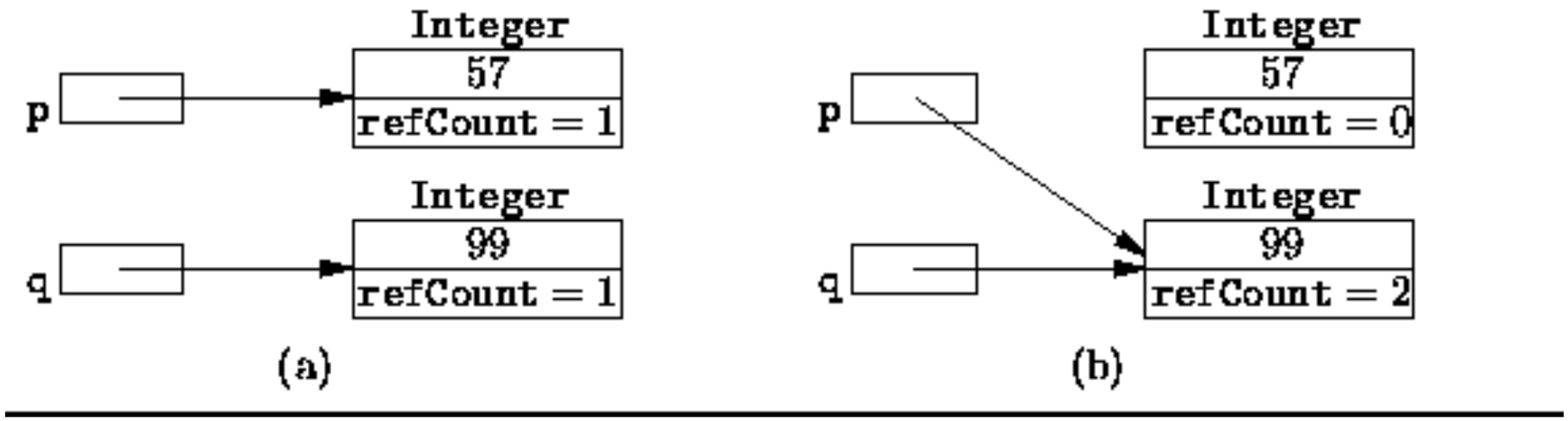
```
Object p = new Integer (57);
```




```
Object p = new Integer (57);
Object q = p;
```

Decrease Counter on Overwrite

```
Object p = new Integer (57);
Object q = new Integer (99);
p = q;
```




```
if (p != q) {
  if (p != null)
    --p.refCount;
  p = q;
  if (p != null)
    ++p.refCount;
}
```

Collect on Decrement to Zero

```
if (p != q) {
  if (p != null)
  if (--p.refCount == 0)
    heap.release (p);
  p = q;
  if (p != null)
    ++p.refCount;
}
```

Assignment: Implement Reference Counting Instrumentation

- Manually inserting code to count references is tedious and error-prone
- Code generator for higher-level language can do this automatically

Implement in BufferC code generator

- Basic idea on previous slides
- But how about
  - variable declarations
  - passing buffers to functions
  - returning buffers from functions
  - buffers referred to by local variables
  - etc.

Bonus: Format String Vulnerabilities

`printf(fmt, s1, …, s2)`

- Takes format string and variable number of string arguments
- Is vulnerable to user generated (or buggy) format string
- Check that format string placeholders match with arguments of printf

Dynamic checking

- Scan the format string
- Generate an implementation of printf for each call site

Static checking

- When format string is a literal: `printf("hello %s", msg)`
- Check number and types of arguments in type system

Super Bonus: Optimization

- Dynamic checking is expensive
- If we know statically that a buffer access is within bounds, we can leave out the dynamic check
- If we can see all the uses of allocated memory, we can release that memory explicitly.
- (This is what C programmers do manually, but it is easy to make mistakes, which leads to vulnerabilities.)
- Based on static (data-flow) analysis, a compiler can make such optimizations safely (but perhaps to conservatively)
- Feel like a challenge? Optimize bounds checks or reference counting

Research

A Language Designer's Workbench

A One-Stop-Shop for Implementation and Verification of Language Designs

Eelco Visser
Delft University of Technology
visser@acm.org

Guido Wachsmuth
Delft University of Technology
guwac@acm.org

Andrew Tolmach
Portland State University
apt@cs.pdx.edu

Pierre Neron, Vlad Vergu
Delft University of Technology
{p.j.m.neron, v.a.vergu}@tudelft.nl

Augusto Passalaqua, Gabriël Konat
Delft University of Technology
{a.passalaquamartins, g.d.p.konat}@tudelft.nl

Abstract

The realization of a language design requires multiple artifacts that redundantly encode the same information. This entails significant effort for language implementors, and often results in late detection of errors in language definitions. In this paper we present a proof-of-concept language designer's workbench that supports generation of IDEs, interpreters, and verification infrastructure from a single source. This constitutes a first milestone on the way to a system that fully automates language implementation and verification.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language classifications; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.4 [*Programming Languages*]: Processors; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

Keywords Language Designer Workbench; Meta-Theory; Language Specification; Syntax; Name Binding; Types; Semantics; Domain Specific Languages

modern software systems. Programming language designers want only one thing: to get usable, reliable realizations of their languages into the hands of programmers as efficiently as possible. To achieve this goal, they need to produce a number of artifacts:

- A compiler or interpreter that allows programmers to execute programs in the language;
- An IDE that supports programmers in constructing programs in the language;
- A high-level specification of the language that documents its intent for programmers;
- Validation, via automated testing or formal verification, that their language designs and implementations are correct and consistent.

Today's savvy language designer knows that there are good tools available to help with these tasks. However, existing tools generally require the designer to create each of these artifacts separately, even though they all reflect the same underlying design. Consequently, a compiler or interpreter is often the only artifact produced; documentation, IDE, and—

What to Submit

Zip with your complete Spoofax project, including

- buffer.h, buffer.c: run-time system
- libstring.bc: library of string functions written in BufferC
- Code generation rules for buffer extension
- Code generation rules for reference count instrumentation
- Test programs for good and bad behavior
- README with your report

Assessment

- Level 1: Correct code generation with bounds access checking (for a 5)
- Level 2: Reference counting (for an 8)
- Level 3: Format string validation (for a 10)
- Level 4: Optimization (for a 10+)

TU Delft

Next: Homework (!) Exam

- Wednesday, January 27, 2016, from 14:00 to17:00 on WebLab
- It seems that there are no facilities for digital exams in Twente. Furthermore, it turns out that in Delft no computer room has been booked yet for the exam.
- So, I'm changing the exam into a ***homework exam***. This means that the exam will still be taken as a digital WebLab exam at the designated time slot (January 27, 14:00-17:00). However, it will not be done in closed room and isolated web environment.
- You should make the exam alone. You should formulate your own answers. Copying from other students, internet, or books is not allowed.
- In case of suspicion of plagiarism, I will conduct an additional oral examination.

TU Delft