

Static Program Analysis Using Type and Effect Systems

Mikhail Belyaev

SPBSPU

11/08/2011

Static Program Analysis

Static Program Analysis

An efficient way to predict runtime behaviors of programs, including software defects.

Typical approaches

- Data flow analysis
- Abstract interpretation
- Inference-based analysis

Static Program Analysis

Static Program Analysis

An efficient way to predict runtime behaviors of programs, including software defects.

Typical approaches

- Inference-based analysis
 - **Type and effect systems:**
 - Simplicity
 - Effectiveness
 - Correctness can be formally proved

Publications

- Jens Palsberg, UCLA — Type Based Analysis and Applications
<http://www.cs.ucla.edu/~palsberg/tba/>
- Flemming Nielson, Hanne R. Nielson, Chris Hankin — Principles of Program Analysis
- Bjarne Steensgaard, MSR — Points-to Analysis in Almost Linear Time
- Chandrasekhar Boyapati and Martin Rinard — A Parameterized Type System for Race-Free Java Programs

and 50+ articles more, mostly theoretical...

Type Systems (Type Theory)

Typing relation

$$\Gamma \vdash e : \tau$$

Where:

Γ is the typing environment

e is the expression

τ is the type

Type Soundness

«Well typed programs do not go wrong» ^a

^aBenjamin C. Pierce, Types and Programming Languages

Type Systems (Type Theory)

Example

$$e ::= v \mid e + e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{isZero } e$$
$$v ::= \text{True} \mid \text{False} \mid 0 \mid 1 \mid \dots$$
$$\tau ::= \text{Bool} \mid \text{Int}$$

Type Systems (Type Theory)

Example

$$e ::= v \mid e + e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{isZero } e$$
$$v ::= \text{True} \mid \text{False} \mid 0 \mid 1 \mid \dots$$
$$\tau ::= \text{Bool} \mid \text{Int}$$
$$\Gamma \vdash \text{True} : \text{Bool} \quad \Gamma \vdash \text{False} : \text{Bool} \quad \Gamma \vdash 0 : \text{Int} \quad \dots$$
$$\frac{\Gamma \vdash x_1 : \text{Int} \quad \Gamma \vdash x_2 : \text{Int}}{\Gamma \vdash (x_1 + x_2) : \text{Int}}$$
$$\frac{\Gamma \vdash x : \text{Int}}{\Gamma \vdash (\text{isZero } x) : \text{Bool}}$$
$$\frac{\Gamma \vdash c : \text{Bool} \quad \Gamma \vdash x_1 : \tau \quad \Gamma \vdash x_2 : \tau}{\Gamma \vdash (\text{if } c \text{ then } x_1 \text{ else } x_2) : \tau}$$

Type Systems for Static Analysis

Type system:

- Target language
- Types
- A set of inference rules

Typing relation

$$\Gamma \vdash e : \tau$$

Type Completeness

If a statement e has a type (i.e. is valid) in the analyzed language, then it should be also valid in the target language.

Example

Target Language

$e ::= \text{Const} \mid \text{Create} \mid \text{Free } e \mid \bigcap e_1 e_2 \mid \text{Return } e \mid e_1 := e_2$

Types

$\tau ::= \text{def} \mid \text{undef}$

Example

Inference rules

$$\begin{array}{c} \Gamma \vdash \text{Const} : \text{def} \\ \Gamma \vdash \text{Create} : \text{undef} \\ \Gamma \vdash \text{Free } e : \text{undef} \\ \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 := e_2 : \tau_2} \\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau = \text{ld}(\tau_1, \tau_2)}{\Gamma \vdash \bigcap e_1 e_2 : \tau} \\ \frac{\Gamma \vdash e : \text{def}}{\Gamma \vdash \text{Return } e : \text{undef}} \end{array}$$

Partial Ordering on Types and Subtyping

- Partial Ordering — a binary relation:

$$\beta \sqsupseteq \alpha$$

or

$$\beta \geq \alpha$$

or even

$$\beta \supseteq \alpha$$

Partial Ordering on Types and Subtyping

- Partial Ordering — a binary relation:

$$\beta \sqsupseteq \alpha$$

or

$$\beta \geq \alpha$$

or even

$$\beta \supseteq \alpha$$

- α is a **subtype** of β

Partial Ordering on Types and Subtyping

- Partial Ordering — a binary relation:

$$\beta \supseteq \alpha$$

- α is a **subtype** of β
- More informal explanations:
 - An α can be supplied wherever a β is needed (contravariant positions)

Partial Ordering on Types and Subtyping

- Partial Ordering — a binary relation:

$$\beta \supseteq \alpha$$

- α is a **subtype** of β
- More informal explanations:
 - An α can be supplied wherever a β is needed (contravariant positions)
 - An α can be considered a β if needed to (covariant positions)

Partial Ordering on Types and Subtyping

- Partial Ordering — a binary relation:

$$\beta \supseteq \alpha$$

- α is a **subtype** of β
- More informal explanations:
 - An α can be supplied wherever a β is needed (contravariant positions)
 - An α can be considered a β if needed to (covariant positions)
 - An α is a β (but not vice versa!)

Partial Ordering on Types and Subtyping

- Partial Ordering — a binary relation:

$$\beta \supseteq \alpha$$

- α is a **subtype** of β
- More informal explanations:
 - An α can be supplied wherever a β is needed (contravariant positions)
 - An α can be considered a β if needed to (covariant positions)
 - An α is a β (but not vice versa!)
- Also in Java:
 - `class B \supseteq class A \iff A extends B`

Partial Ordering on Types and Subtyping: lattices

- A semilattice is a partially ordered set that has a single greatest or least element

Partial Ordering on Types and Subtyping: lattices

- A semilattice is a partially ordered set that has a single greatest or least element
- A lattice (complete lattice) is a partially ordered set that has a single greatest **and** a single least element

Partial Ordering on Types and Subtyping: lattices

- A semilattice is a partially ordered set that has a single greatest or least element
- A lattice (complete lattice) is a partially ordered set that has a single greatest **and** a single least element
- Occasional notation for types: \top and \perp

Partial Ordering on Types and Subtyping: lattices

- A semilattice is a partially ordered set that has a single greatest or least element
- A lattice (complete lattice) is a partially ordered set that has a single greatest **and** a single least element
- Occasional notation for types: \top and \perp
- Also in Java:

Partial Ordering on Types and Subtyping: lattices

- A semilattice is a partially ordered set that has a single greatest or least element
- A lattice (complete lattice) is a partially ordered set that has a single greatest **and** a single least element
- Occasional notation for types: \top and \perp
- Also in Java:
 - The «reference» part of typesystem is a complete lattice

Partial Ordering on Types and Subtyping: lattices

- A semilattice is a partially ordered set that has a single greatest or least element
- A lattice (complete lattice) is a partially ordered set that has a single greatest **and** a single least element
- Occasional notation for types: \top and \perp
- Also in Java:
 - The «reference» part of typesystem is a complete lattice
 - The top element is `Object`

Partial Ordering on Types and Subtyping: lattices

- A semilattice is a partially ordered set that has a single greatest or least element
- A lattice (complete lattice) is a partially ordered set that has a single greatest **and** a single least element
- Occasional notation for types: \top and \perp
- Also in Java:
 - The «reference» part of typesystem is a complete lattice
 - The top element is `Object`
 - The bottom element is `null`

Partial Ordering on Types and Subtyping: lattices

- A semilattice is a partially ordered set that has a single greatest or least element
- A lattice (complete lattice) is a partially ordered set that has a single greatest **and** a single least element
- Occasional notation for types: \top and \perp
- Also in Java:
 - The «reference» part of typesystem is a complete lattice
 - The top element is `Object`
 - The bottom element is `null`
- And in Scala:

Partial Ordering on Types and Subtyping: lattices

- A semilattice is a partially ordered set that has a single greatest or least element
- A lattice (complete lattice) is a partially ordered set that has a single greatest **and** a single least element
- Occasional notation for types: \top and \perp
- Also in Java:
 - The «reference» part of typesystem is a complete lattice
 - The top element is `Object`
 - The bottom element is `null`
- And in Scala:
 - The whole typesystem is a lattice

Partial Ordering on Types and Subtyping: lattices

- A semilattice is a partially ordered set that has a single greatest or least element
- A lattice (complete lattice) is a partially ordered set that has a single greatest **and** a single least element
- Occasional notation for types: \top and \perp
- Also in Java:
 - The «reference» part of typesystem is a complete lattice
 - The top element is `Object`
 - The bottom element is `null`
- And in Scala:
 - The whole typesystem is a lattice
 - The top element is `Any`

Partial Ordering on Types and Subtyping: lattices

- A semilattice is a partially ordered set that has a single greatest or least element
- A lattice (complete lattice) is a partially ordered set that has a single greatest **and** a single least element
- Occasional notation for types: \top and \perp
- Also in Java:
 - The «reference» part of typesystem is a complete lattice
 - The top element is `Object`
 - The bottom element is `null`
- And in Scala:
 - The whole typesystem is a lattice
 - The top element is `Any`
 - The bottom element is `Nothing`

Type Systems for Static Analysis

Type system:

- Target language
- Types
- A set of inference rules
- The partial ordering on types

Questions

- Does the type system have to be a complete lattice?

Type Systems for Static Analysis

Type system:

- Target language
- Types
- A set of inference rules
- **The partial ordering on types**

Questions

- Does the type system have to be a complete lattice?
- ... or an upper semilattice?

Type Systems for Static Analysis

Type system:

- Target language
- Types
- A set of inference rules
- **The partial ordering on types**

Questions

- Does the type system have to be a complete lattice?
- ... or an upper semilattice?
- ... or a lower semilattice?

Example: subtypes

Target language

$e ::= \text{Const} \mid \text{Create} \mid \text{Free } e \mid \bigcup e_1 e_2 \mid \bigcap e_1 e_2 \mid \text{Return } e \mid e_1 := e_2$

Types

$\tau ::= \text{def} \mid \text{undef} \mid \text{maydef}$

$\text{maydef} \sqsubseteq \text{def} \quad \text{maydef} \sqsubseteq \text{undef}$

Example: subtypes

Inference rules

$$\Gamma \vdash \text{Const} : \text{def}$$
$$\Gamma \vdash \text{Create} : \text{undef}$$
$$\Gamma \vdash \text{Free } e : \text{undef}$$
$$\frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 := e_2 : \tau_2}$$
$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau \sqsupseteq \tau_1 \quad \tau \sqsupseteq \tau_2}{\Gamma \vdash \bigcup e_1 e_2 : \tau}$$
$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau = \text{ld}(\tau_1, \tau_2)}{\Gamma \vdash \bigcap e_1 e_2 : \tau}$$
$$\Gamma \vdash e : \text{def}$$
$$\frac{\Gamma \vdash e : \text{def}}{\Gamma \vdash \text{Return } e : \text{undef}}$$

What do we have now?

- An analysis that actually works

What do we have now?

- An analysis that actually works
- Detection of single defect

What do we have now?

- An analysis that actually works
- Detection of single defect
- What if the defect is a false one?

Annotations on types

$\tau ::= \text{def} \mid \text{undef} \mid \text{maydef}$

Annotations on types

$$\tau ::= \text{def} \mid \text{undef} \mid \text{maydef}$$

May be expressed as:

$$\tau ::= \psi_{\varphi}$$

ψ is the «natural» type

$$\varphi ::= \text{def} \mid \text{undef} \mid \text{maydef}$$

*bool*_{def}, *int*_{undef}, etc.

Annotations on types

$$\tau ::= \text{def} \mid \text{undef} \mid \text{maydef}$$

May be expressed as:

$$\tau ::= \psi_\varphi$$

ψ is the «natural» type

$$\varphi ::= \text{def} \mid \text{undef} \mid \text{maydef}$$

*bool*_{def}, *int*_{undef}, etc.

Very much like C qualifiers (CQual): `def bool, undef int`

Annotations on types

$$\tau ::= \text{def} \mid \text{undef} \mid \text{maydef}$$

May be expressed as:

$$\tau ::= \psi_\varphi$$

ψ is the «natural» type

$$\varphi ::= \text{def} \mid \text{undef} \mid \text{maydef}$$

*bool*_{def}, *int*_{undef}, etc.

Very much like C qualifiers (CQual): `def bool, undef int`
We also may put annotations on top of existent analysis.

Type and Effect Systems

Type and Effect System:

- Target Language
- Types
- A set of inference rules (for both types and effects)
- The partial ordering on types

Typing relation

$$\Gamma \vdash e : \tau \& \varphi$$

Where:

Γ is the type environment

e is the expression

τ is the type

φ is the set of effects

Example

Target language

$$e ::= \text{Const} \mid \text{Create} \mid \text{Free } e \mid \bigcup e_1 e_2 \mid \bigcap e_1 e_2 \mid \text{Return } e \mid e_1 := e_2$$

Types

$$\tau ::= \text{def} \mid \text{undef} \mid \text{maydef}$$
$$\text{maydef} \sqsupseteq \text{def} \quad \text{maydef} \sqsupseteq \text{undef}$$

Example

Inference rules

$$\begin{array}{c} \Gamma \vdash \text{Const} : \text{def} \ \& \ \{\} \quad \Gamma \vdash \text{Create} : \text{undef} \ \& \ \{\} \\ \Gamma \vdash \text{Free } e : \text{undef} \ \& \ \{\} \quad \frac{\Gamma \vdash e_1 : \tau_1 \ \& \ \varphi_1 \quad \Gamma \vdash e_2 : \tau_2 \ \& \ \varphi_2}{\Gamma \vdash e_1 := e_2 : \tau_2 \ \& \ (\varphi_1 \cup \varphi_2)} \\ \frac{\Gamma \vdash e_1 : \tau_1 \ \& \ \varphi_1 \quad \Gamma \vdash e_2 : \tau_2 \ \& \ \varphi_2 \quad \tau \sqsupseteq \tau_1 \quad \tau \sqsupseteq \tau_2}{\Gamma \vdash \bigcup e_1 e_2 : \tau \ \& \ (\varphi_1 \cup \varphi_2)} \\ \frac{\Gamma \vdash e_1 : \tau_1 \ \& \ \varphi_1 \quad \Gamma \vdash e_2 : \tau_2 \ \& \ \varphi_2 \quad \tau = \text{ld}(\tau_1, \tau_2)}{\Gamma \vdash \bigcap e_1 e_2 : \tau \ \& \ (\varphi_1 \cup \varphi_2)} \\ \frac{\Gamma \vdash e : \text{def} \ \& \ \varphi_0}{\Gamma \vdash \text{Return } e : \text{undef} \ \& \ \varphi_0} \\ \frac{\Gamma \vdash e : \text{undef} \ \& \ \varphi_0}{\Gamma \vdash \text{Return } e : \text{undef} \ \& \ (\varphi_0 \cup \{\text{ERR}\})} \\ \frac{\Gamma \vdash e : \text{maydef} \ \& \ \varphi_0}{\Gamma \vdash \text{Return } e : \text{undef} \ \& \ (\varphi_0 \cup \{\text{ERR}\})} \end{array}$$

Getting more complicated...

- Special polymorphism
 - Type families $\forall a.a \rightarrow a$
 - Existential types
- Complex effects (depending on types)
- Subeffecting
- etc.

Using types for program analysis

- Types as discriminators
 - Class Hierarchy Analysis, Rapid Type Analysis, etc.
 - Swift compiler, JAX optimiser, etc.
- Type and effect systems
 - Concurrent Java
 - Large theoretical base
 - ?

The idea

Apply type and effect systems to real programs written in C

Choosing program model

- Designing our own model
- Modifying an existent one
 - AST — abstract syntax tree
 - ASG — abstract semantic graph
 - DDG — data dependency graph
 - SSA — static single assignment form

Choosing program model

- Designing our own model
- Modifying an existent one
 - AST — abstract syntax tree
 - ASG — abstract semantic graph
 - DDG — data dependency graph
 - **SSA** — static single assignment form

Data dependency info

Phi-functions

Retrieving the SSA

Some facts

LLVM compiler infrastructure uses SSA as an intermediate representation. LLVM frontends (clang, llvm-gcc, etc.) can emit LLVM IR.

Retrieving the SSA

Some facts

LLVM compiler infrastructure uses SSA as an intermediate representation. LLVM frontends (clang, llvm-gcc, etc.) can emit LLVM IR.

llvm-parser

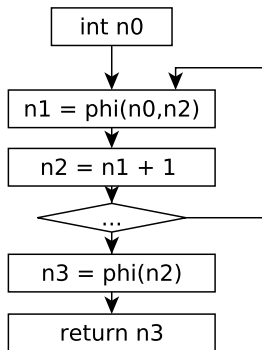
- Standalone parser of LLVM bitcode files
- Retrieves the model from files and makes it more suitable for typing
- Keeps the metadata info

How this works

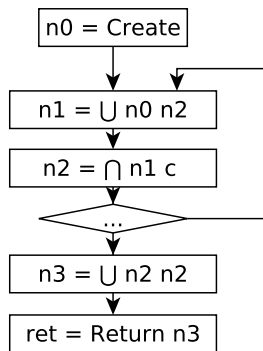
C

```
int n;  
do {  
    n++;  
} while (...);  
return n;
```

⇒ SSA



⇒ Target language

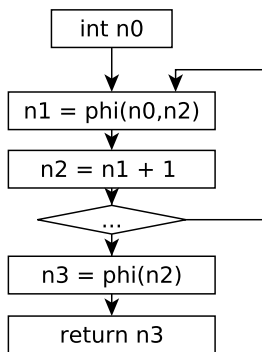


How this works

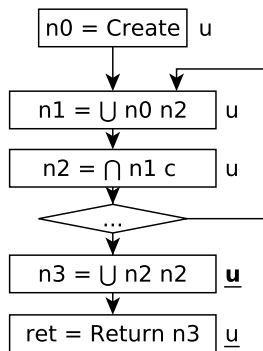
C

```
int n;  
do {  
    n++;  
} while (...);  
return n;
```

⇒ SSA



⇒ Target language



ERR!

Solving type ambiguity

Using SMT-solver

- 1 Convert rules to first-order logic theorems
- 2 Find the correct solution with the minimal number of effects

Drawbacks

No guarantees and need to encode everything (types, effects, rules, etc.)

Tools — SBV library and its EDSL:

```
rules Constant = definition DDefined
rules (Assign val exp) =
  λ x e s → let t = s ⊢ exp in x ≡ t
rules (JoinOr v0 v1) =
  λ x e s → let t0 = s ⊢ v0; t1 = s ⊢ v1 in
    x 'pge' t0 ∧ x 'pge' t1
rules (Return v) =
  λ x e s → let t = s ⊢ v in
    if_ (DUndefined 'typeEq' t ∨ DMaybeDefined 'typeEq' t)
      (e 'effect' 0)
```

¹<https://github.com/LeventErkok/sbv>

And in C?

- There are pointers in C

And in C?

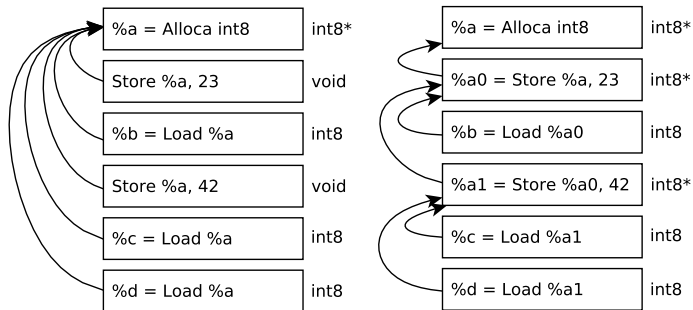
- There are pointers in C
- There also are pointers to pointers to pointers...

And in C?

- There are pointers in C
- There also are pointers to pointers to pointers...

What to do?

Pointer versions!



And in C?

- There are pointers in C
- There also are pointers to pointers to pointers...

What to do?

Pointer versions!

How to make it?

- Stores and Loads
- Which instructions point to the same memory cell?

And in C?

- There are pointers in C
- There also are pointers to pointers to pointers...

What to do?

Pointer versions!

How to make it?

- Stores and Loads LLVM IR already contains them!
- Which instructions point to the same memory cell? need pointer analysis

Pointer analysis

- Alias Analysis — does the pointer A point to the same location as pointer B does?
- Point-to Analysis — model the heap as some kind of graph

Pointer analysis

- Alias Analysis — does the pointer A point to the same location as pointer B does?
- Point-to Analysis — model the heap as some kind of graph

What's the difference?

- It's almost no difference really

Pointer analysis

- Alias Analysis — does the pointer A point to the same location as pointer B does?
- Point-to Analysis — model the heap as some kind of graph

What's the difference?

- It's almost no difference really
- Flow-insensitive analysis is NP-complete :)

Pointer analysis

- Alias Analysis — does the pointer A point to the same location as pointer B does?
- Point-to Analysis — model the heap as some kind of graph

What's the difference?

- It's almost no difference really
- Flow-insensitive analysis is NP-complete :)
- Flow-sensitive analysis is undecidable :)

Pointer analysis

- Alias Analysis — does the pointer A point to the same location as pointer B does?
- Point-to Analysis — model the heap as some kind of graph

What's the difference?

- It's almost no difference really
- Flow-insensitive analysis is NP-complete :)
- Flow-sensitive analysis is undecidable :)
- Our project provides a very naive implementation of AA

Problems

The approach

- Need more precise pointer analysis
- Interprocedural analysis only
- Globals have no special treatment

The prototype

- No analysis of conditions
- No variable arguments support

Plans

- Implement more precise pointer analysis
 - ...As a type and effect system maybe?
- Try out more type and effect systems
- Incorporate intraprocedural analysis through function constraints
- Add support for more complex systems:
 - Subeffecting
 - Dependency on conditions
- Usage for purposes other than defect detection

Links

- LLVM Parser — <http://llvm-parser.googlecode.com>
- LLVM Analyzer —
<http://tiger.ftk.spbstu.ru/trac/llvm-analyzer>

QUESTIONS?