Static Program Analysis Using Type and Effect Systems

Mikhail Belyaev

SPBSPU

27 марта 2012 г.

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]

[a] Benjamin C. Pierce, Types and Programming Languages

Type Systems (Type Theory)

Example

$$e ::= v \mid e + e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{isZero } e$$

$$v ::= \text{True} \mid \text{False} \mid 0 \mid 1 \mid \cdots$$

$$\tau ::= \text{Bool} \mid \text{Int}$$

Type Systems (Type Theory)

Example

$$e ::= v \mid e + e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{isZero } e$$

$$v ::= \text{True} \mid \text{False} \mid 0 \mid 1 \mid \cdots$$

$$\tau ::= \text{Bool} \mid \text{Int}$$

$$\Gamma \vdash \text{True} : \text{Bool} \quad \Gamma \vdash \text{False} : \text{Bool} \quad \Gamma \vdash 0 : \text{Int} \quad \cdots$$

$$\frac{\Gamma \vdash x_1 : \text{Int} \quad \Gamma \vdash x_2 : \text{Int}}{\Gamma \vdash (x_1 + x_2) : \text{Int}}$$

$$\frac{\Gamma \vdash x : \text{Int}}{\Gamma \vdash (\text{isZero } x) : \text{Bool}}$$

$$\frac{\Gamma \vdash c : \text{Bool} \quad \Gamma \vdash x_1 : \tau \quad \Gamma \vdash x_2 : \tau}{\Gamma \vdash (\text{if } c \text{ then } x_1 \text{ else } x_2) : \tau}$$

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

$$\Gamma \vdash \mathsf{Const} : \mathsf{def}$$

$$\Gamma \vdash \mathsf{Create} : \mathsf{undef}$$

$$\Gamma \vdash \mathsf{Free}\ e : \mathsf{undef}$$

$$\frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 := e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau = \mathtt{ld}(\tau_1, \tau_2)}{\Gamma \vdash \bigcap e_1\, e_2 : \tau}$$

$$\frac{\Gamma \vdash e : \mathsf{def}}{\Gamma \vdash \mathsf{Return}\ e : \mathsf{undef}}$$

Partial Ordering on Types and Subtyping

- Partial Ordering — a binary relation:

$$\beta \sqsupseteq \alpha$$

Partial Ordering on Types and Subtyping

- Partial Ordering — a binary relation:

$$\beta \sqsupseteq \alpha$$

or

$$\beta \geq \alpha$$

or

$$\beta >: \alpha$$

or even

$$\beta \trianglerighteq \alpha$$

Partial Ordering on Types and Subtyping

- Partial Ordering — a binary relation:

$$\beta \sqsupseteq \alpha$$

- 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:

$$\beta \sqsupseteq \alpha$$

- 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:

$$\beta \sqsupseteq \alpha$$

- 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:

$$\beta \sqsupseteq \alpha$$

- 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 >: class A <=> 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 $\bot$

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 $\bot$
- 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 $\bot$
- 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 $\bot$
- 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 $\bot$
- 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 $\bot$
- 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 $\bot$
- 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 $\bot$
- 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 $\bot$
- 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

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

$$\text{maydef} \sqsupseteq \text{def} \qquad \text{maydef} \sqsupseteq \text{undef}$$

Example: subtypes

Inference rules

$$\Gamma \vdash \mathsf{Const} : \mathsf{def}$$

$$\Gamma \vdash \mathsf{Create} : \mathsf{undef}$$

$$\Gamma \vdash \mathsf{Free}\ e : \mathsf{undef}$$

$$\frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 := e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau \sqsupseteq \tau_1 \quad \tau \sqsupseteq \tau_2}{\Gamma \vdash \bigcup e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau = \mathtt{ld}(\tau_1,\ \tau_2)}{\Gamma \vdash \bigcap e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e : \mathsf{def}}{\Gamma \vdash \mathsf{Return}\ e : \mathsf{undef}}$$

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

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

Annotations on types

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

May be expressed as:

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

$$\psi \text{ is the «natural» type}$$

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

$$\mathit{bool}_{\text{def}}, \quad \mathit{int}_{\text{undef}}, \quad \mathit{etc}.$$

Annotations on types

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

May be expressed as:

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

$$\psi \text{ is the «natural» type}$$

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

$$bool_{\text{def}}, \quad int_{\text{undef}}, \quad etc.$$

C qualifiers (CQUAL)

`def bool, undef int`

Annotations on types

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

May be expressed as:

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

$$\psi \text{ is the «natural» type}$$

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

$$bool_{\text{def}}, \quad int_{\text{undef}}, \quad etc.$$

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

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

$$\text{maydef} \sqsupseteq \text{def} \qquad \text{maydef} \sqsupseteq \text{undef}$$

Example

Inference rules

$$\Gamma \vdash \text{Const} : \text{def} \,\&\, \{\} \qquad \Gamma \vdash \text{Create} : \text{undef} \,\&\, \{\}$$

$$\Gamma \vdash \text{Free } e : \text{undef} \,\&\, \{\} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \& \varphi_1 \quad \Gamma \vdash e_2 : \tau_2 \& \varphi_2}{\Gamma \vdash e_1 := e_2 : \tau_2 \& (\varphi_1 \cup \varphi_2)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \& \varphi_1 \quad \Gamma \vdash e_2 : \tau_2 \& \varphi_2 \quad \tau \sqsupseteq \tau_1 \quad \tau \sqsupseteq \tau_2}{\Gamma \vdash \bigcup e_1\, e_2 : \tau \& (\varphi_1 \cup \varphi_2)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \& \varphi_1 \quad \Gamma \vdash e_2 : \tau_2 \& \varphi_2 \quad \tau = \mathtt{ld}(\tau_1,\ \tau_2)}{\Gamma \vdash \bigcap e_1\, e_2 : \tau \& (\varphi_1 \cup \varphi_2)}$$

$$\frac{\Gamma \vdash e : \text{def} \,\&\, \varphi_0}{\Gamma \vdash \text{Return } e : \text{undef} \,\&\, \varphi_0}$$

$$\frac{\Gamma \vdash e : \text{undef} \,\&\, \varphi_0}{\Gamma \vdash \text{Return } e : \text{undef} \,\&\, (\varphi_0 \cup \{\mathtt{ERR}\})}$$

$$\frac{\Gamma \vdash e : \text{maydef} \,\&\, \varphi_0}{\Gamma \vdash \text{Return } e : \text{undef} \,\&\, (\varphi_0 \cup \{\mathtt{ERR}\})}$$

Getting more complicated...

- Special polymorphism
  - Type families $\forall a.a \to a$
  - Existential types $\exists b.a \to b$
- Complex effects (depending on types)
- Subeffects
- Dependent and recursive types $A = A \to A$
- Subeffecting
- etc.

Liquid Types

Logically Qualified Types

Liquid Types

Logically Qualified Types

Notation

$\{\nu : \tau | p\nu \wedge q\nu \wedge \ldots\}$

Liquid Types

Logically Qualified Types

Notation

$\{\nu : \tau | p\nu \wedge q\nu \wedge \ldots\}$

$\{\nu : Int | \nu > 0 \wedge \nu < 2\}$

Liquid Types

Logically Qualified Types

Notation

$\{\nu : \tau | p\nu \wedge q\nu \wedge \ldots\}$

$\{\nu : Int | \nu > 0 \wedge \nu < 2\}$
$\tau = \{\nu : \tau | true\}$

Liquid Types

Logically Qualified Types

Notation

$\{\nu : \tau | p\nu \wedge q\nu \wedge \ldots\}$

$\{\nu : Int | \nu > 0 \wedge \nu < 2\}$
$\tau = \{\nu : \tau | true\}$
$\{\nu_1 : \tau_1 | \nu_1 > 0\} \rightarrow \{\nu_2 : \tau_2 | \nu_2 > \nu_1\}$

Liquid Types

Logically Qualified Types

Notation

$\{\nu : \tau | p\nu \wedge q\nu \wedge \ldots\}$

$\{\nu : Int | \nu > 0 \wedge \nu < 2\}$
$\tau = \{\nu : \tau | true\}$
$\{\nu_1 : \tau_1 | \nu_1 > 0\} \rightarrow \{\nu_2 : \tau_2 | \nu_2 > \nu_1\}$

Subtyping

$$\{\nu : \tau | X(\nu)\} \sqsupseteq \{\nu : \tau | Y(\nu)\}$$
$$=$$
$$\forall \nu : X(\nu) \Rightarrow Y(\nu)$$

Problem #1: Language

- Intermediate language is in every way functional

Problem #1: Language

- Intermediate language is in every way functional
- We are interested in complex procedural languages

Problem #1: Language

- Intermediate language is in every way functional
- We are interested in complex procedural languages
- Syntax-based approach becomes not so syntax-based...

Problem #1: Language

- Intermediate language is in every way functional
- We are interested in complex procedural languages
- Syntax-based approach becomes not so syntax-based...

Problem #1: Language

- Intermediate language is in every way functional
- We are interested in complex procedural languages
- Syntax-based approach becomes not so syntax-based...

What to do?

Problem #1: Language

- Intermediate language is in every way functional
- We are interested in complex procedural languages
- Syntax-based approach becomes not so syntax-based...

What to do?

- Make it functional!

Problem #1: Language

- Intermediate language is in every way functional
- We are interested in complex procedural languages
- Syntax-based approach becomes not so syntax-based...

What to do?

- Make it functional!
- At least, kinda'

Problem #1: Language

- Intermediate language is in every way functional
- We are interested in complex procedural languages
- Syntax-based approach becomes not so syntax-based...

What to do?

- Make it functional!
- At least, kinda'
- Well, as functional as we can :-(

Problem #2: Inference algorithm

Substitution & Unification

- We have an equation: $X = Y$

Problem #2: Inference algorithm

Substitution & Unification

- We have an equation: $X = Y$
- X and Y both contain variables

Problem #2: Inference algorithm

Substitution & Unification

- We have an equation: $X = Y$
- X and Y both contain variables
- We substitute (replace) all vars in X and all vars in Y in such way that $X = Y$ becomes obvious

Problem #2: Inference algorithm

Substitution & Unification

- We have an equation: $X = Y$
- X and Y both contain variables
- We substitute (replace) all vars in X and all vars in Y in such way that $X = Y$ becomes obvious
- PROFIT

Problem #2: Inference algorithm

Substitution & Unification

- We have an equation: $X = Y$
- X and Y both contain variables
- We substitute (replace) all vars in X and all vars in Y in such way that $X = Y$ becomes obvious
- PROFIT

Constraints & Solving

- We have many equations: $X_1 = Y_1 \quad X_2 = Y_2 \quad \ldots$

Problem #2: Inference algorithm

Substitution & Unification

- We have an equation: $X = Y$
- X and Y both contain variables
- We substitute (replace) all vars in X and all vars in Y in such way that $X = Y$ becomes obvious
- PROFIT

Constraints & Solving

- We have many equations: $X_1 = Y_1 \quad X_2 = Y_2 \quad \ldots$
- We construct constraints on all variables: $px_K = 0$

Problem #2: Inference algorithm

Substitution & Unification

- We have an equation: $X = Y$
- X and Y both contain variables
- We substitute (replace) all vars in X and all vars in Y in such way that $X = Y$ becomes obvious
- PROFIT

Constraints & Solving

- We have many equations: $X_1 = Y_1 \quad X_2 = Y_2 \quad \ldots$
- We construct constraints on all variables: $px_K = 0$
- Connect all the constraints with a $\wedge$

Problem #2: Inference algorithm

Substitution & Unification

- We have an equation: $X = Y$
- X and Y both contain variables
- We substitute (replace) all vars in X and all vars in Y in such way that $X = Y$ becomes obvious
- PROFIT

Constraints & Solving

- We have many equations: $X_1 = Y_1 \quad X_2 = Y_2 \quad \ldots$
- We construct constraints on all variables: $px_K = 0$
- Connect all the constraints with a $\wedge$
- Solve the HUGE equation

Problem #2: Inference algorithm

Substitution & Unification

- We have an equation: $X = Y$
- X and Y both contain variables
- We substitute (replace) all vars in X and all vars in Y in such way that $X = Y$ becomes obvious
- PROFIT

Constraints & Solving

- We have many equations: $X_1 = Y_1 \quad X_2 = Y_2 \quad \ldots$
- We construct constraints on all variables: $px_K = 0$
- Connect all the constraints with a $\wedge$
- Solve the HUGE equation
- PROFIT

Problem #2: Inference algorithm

- Only trivial systems can be inferred easily

Problem #2: Inference algorithm

- Only trivial systems can be inferred easily
- General inference is NP-hard

Problem #2: Inference algorithm

- Only trivial systems can be inferred easily
- General inference is NP-hard
- Subtyping makes answers non-deterministic

Problem #2: Inference algorithm

- Only trivial systems can be inferred easily
- General inference is NP-hard
- Subtyping makes answers non-deterministic
- Combining subtyping with other features makes it even worse

Problem #2: Inference algorithm

- Only trivial systems can be inferred easily
- General inference is NP-hard
- Subtyping makes answers non-deterministic
- Combining subtyping with other features makes it even worse
- Existential and recursive typing make inference undecidable

Problem #2: Inference algorithm

- Only trivial systems can be inferred easily
- General inference is NP-hard
- Subtyping makes answers non-deterministic
- Combining subtyping with other features makes it even worse
- Existential and recursive typing make inference undecidable

What to do?

Problem #2: Inference algorithm

- Only trivial systems can be inferred easily
- General inference is NP-hard
- Subtyping makes answers non-deterministic
- Combining subtyping with other features makes it even worse
- Existential and recursive typing make inference undecidable

What to do?

- There are plenty of algorithms for different cases

Problem #2: Inference algorithm

- Only trivial systems can be inferred easily
- General inference is NP-hard
- Subtyping makes answers non-deterministic
- Combining subtyping with other features makes it even worse
- Existential and recursive typing make inference undecidable

What to do?

- There are plenty of algorithms for different cases
- Some of them are actually more effective than any flow-based approaches

Problem #2: Inference algorithm

- Only trivial systems can be inferred easily
- General inference is NP-hard
- Subtyping makes answers non-deterministic
- Combining subtyping with other features makes it even worse
- Existential and recursive typing make inference undecidable

What to do?

- There are plenty of algorithms for different cases
- Some of them are actually more effective than any flow-based approaches
- We can also transform our (non-deterministic) system to algorithmic (trivial) one

Problem #2: Inference algorithm

- Only trivial systems can be inferred easily
- General inference is NP-hard
- Subtyping makes answers non-deterministic
- Combining subtyping with other features makes it even worse
- Existential and recursive typing make inference undecidable

What to do?

- There are plenty of algorithms for different cases
- Some of them are actually more effective than any flow-based approaches
- We can also transform our (non-deterministic) system to algorithmic (trivial) one
- We can use an external deus-ex-machina, a superman that will save us all...

Problem #2: Inference algorithm

- Only trivial systems can be inferred easily
- General inference is NP-hard
- Subtyping makes answers non-deterministic
- Combining subtyping with other features makes it even worse
- Existential and recursive typing make inference undecidable

What to do?

- There are plenty of algorithms for different cases
- Some of them are actually more effective than any flow-based approaches
- We can also transform our (non-deterministic) system to algorithmic (trivial) one
- We can use an external deus-ex-machina, a superman that will save us all...
  - Theorem provers: HOL, Coq, etc.
  - SMT-solvers: Z3, Alt-Ergo, Yices, etc.

Problem #3: Proving the correctness

- The hardest part is to answer the question
  «What do we actually want to prove?»

Problem #3: Proving the correctness

- The hardest part is to answer the question
  «What do we actually want to prove?»
- The answer is different for every system

Problem #3: Proving the correctness

- The hardest part is to answer the question
  «What do we actually want to prove?»
- The answer is different for every system
- Or is it?

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.

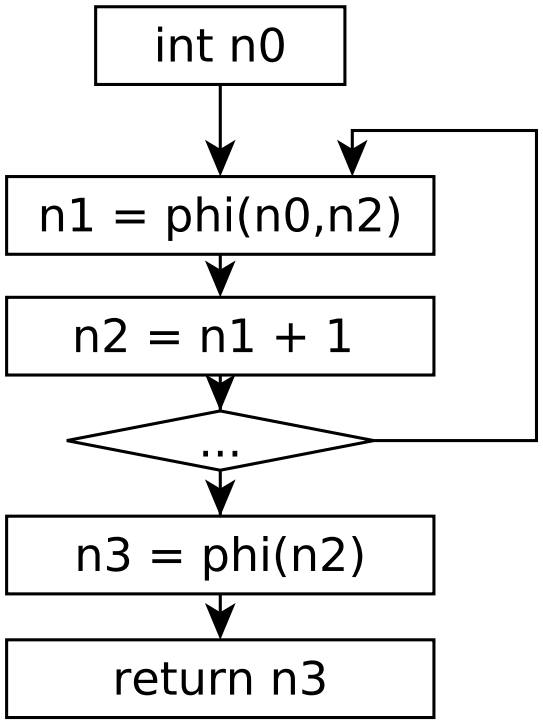
llvm-parser

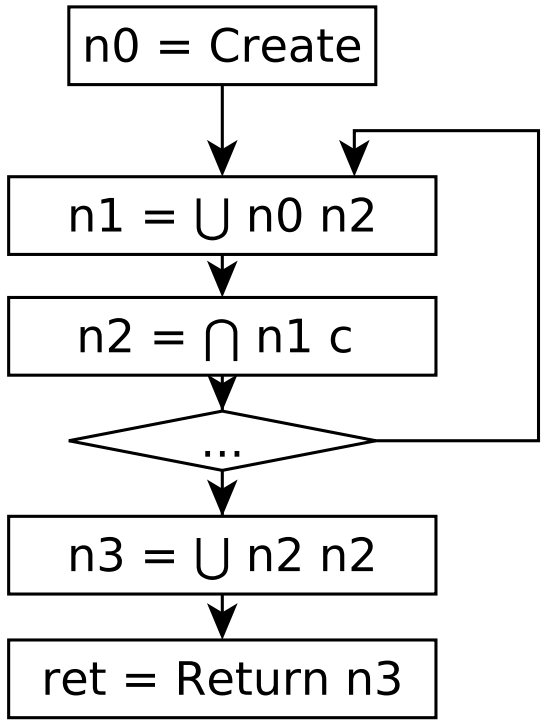
- Standalone parser of LLVM bitcode files
- Retrieves the model from files and makes it more suitable for typing
- Keeps the metadata info

How this works

C ⇒ SSA ⇒ Target language

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

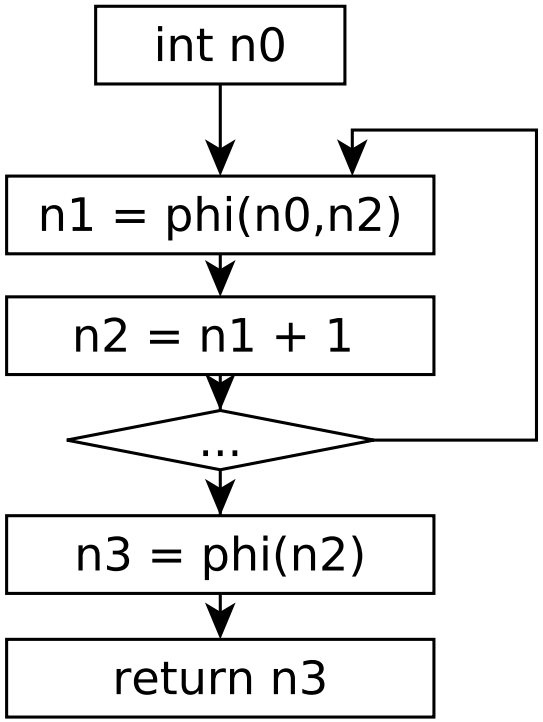


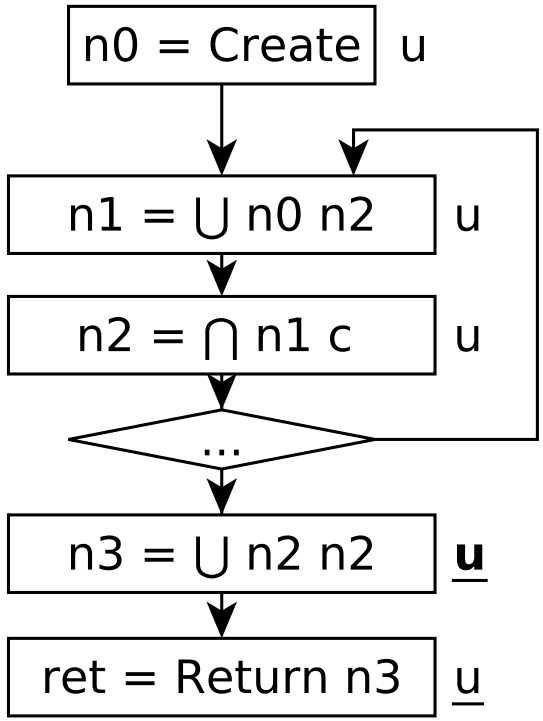


How this works

C ⇒ SSA ⇒ Target language

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





ERR!

SMT-solvers

- Satisfiability Modulo Theories

SMT-solvers

- Satisfiability Modulo Theories
- SAT-solver + T-theory solvers

SMT-solvers

- Satisfiability Modulo Theories
- SAT-solver + T-theory solvers
- SAT:

For a logical formula $X_1X_2X_3 \vee X_4$ what are X_K that evaluate it to true?

SMT-solvers

- Satisfiability Modulo Theories
- SAT-solver + T-theory solvers
- SAT:

For a logical formula $X_1X_2X_3 \vee X_4$ what are X_K that evaluate it to true?

- T-theories: natural numbers, floats, polynomials, etc.

SMT-solvers

- Satisfiability Modulo Theories
- SAT-solver + T-theory solvers
- SAT:

For a logical formula $X_1 X_2 X_3 \vee X_4$ what are X_K that evaluate it to true?

- T-theories: natural numbers, floats, polynomials, etc.
- SMT:

For a mathematical formula $X_1 * X_2 + X_3 = 500 \vee X_4 = 2$ what are X_K that evaluate it to true?

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.)

SBV library[1]

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?

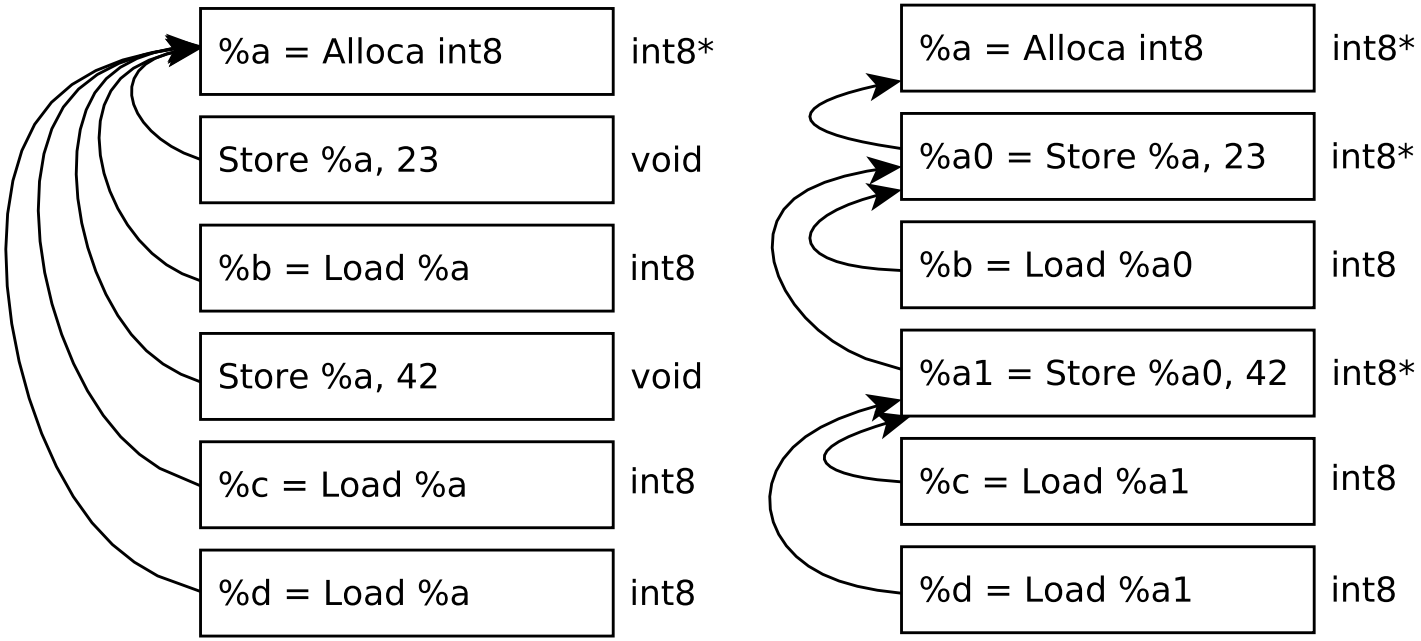
- There are pointers in C
- Type and Effect Systems are flow-insensitive — they don't handle pointers!
- There also are pointers to pointers to pointers...

And in C?

- There are pointers in C
- Type and Effect Systems are flow-insensitive — they don't handle pointers!
- There also are pointers to pointers to pointers...

What to do?

Pointer versions!



And in C?

- There are pointers in C
- Type and Effect Systems are flow-insensitive — they don't handle pointers!
- 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
- Type and Effect Systems are flow-insensitive — they don't handle pointers!
- There also are pointers to pointers to pointers...

What to do?

Pointer versions!

How to make it?

- Stores and Loads — LLVM already has 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
- Context-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
- Context-insensitive analysis is NP-complete :)
- Context-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
- Context-insensitive analysis is NP-complete :)
- Context-sensitive analysis is undecidable :)
- Our project provides a very naive implementation of AA

Problems

- No working general solution :)

THE END