# *Logic Programs as Compact Denotations*

## *PADL'03, New Orleans, January 2003*


Fausto Spoto

Dipartimento di Informatica, Verona, Italy

Patricia M. Hill

School of Computing, Leeds, UK

# *Credits*

Supported by

EPSRC grant “Escape Analysis of Object-Oriented Languages”

# *Credits*

Supported by

EPSRC grant “Escape Analysis of Object-Oriented Languages”

MURST grant “Interpretazione Astratta, Sistemi di Tipo e Analisi Control-Flow”.

# *Our Goal*

Compositional static analysis of object-oriented languages

# *Our Goal*

Compositional static analysis of object-oriented languages

- Compositionality is obtained by using a denotational semantics

# *Our Goal*

Compositional static analysis of object-oriented languages

- Compositionality is obtained by using a denotational semantics
- Denotations are maps. We show how to compact them and make them more efficient

# *Our Goal*

Compositional static analysis of object-oriented languages

- Compositionality is obtained by using a denotational semantics
- Denotations are maps. We show how to compact them and make them more efficient
- This is achieved through their representation in terms of logic programs

# *Our Goal*

Compositional static analysis of object-oriented languages

- Compositionality is obtained by using a denotational semantics
- Denotations are maps. We show how to compact them and make them more efficient
- This is achieved through their representation in terms of logic programs
- Some experiments show the advantages of using our technique even compared to BDD's.

# *Denotational Semantics*

- clean theory

# *Denotational Semantics*

- clean theory
- specified from the denotations of the basic operations of the language

# *Denotational Semantics*



- clean theory
- specified from the denotations of the basic operations of the language
  $\Rightarrow$ assignment

# *Denotational Semantics*



- clean theory
- specified from the denotations of the basic operations of the language
  $\Rightarrow$ assignment
  $\Rightarrow$ scope expansion

# *Denotational Semantics*

- clean theory
- specified from the denotations of the basic operations of the language
  $\Rightarrow$ assignment
  $\Rightarrow$ scope expansion
  $\Rightarrow$ scope restriction

# *Denotational Semantics*



- clean theory
- specified from the denotations of the basic operations of the language
  - $\Rightarrow$ assignment
  - $\Rightarrow$ scope expansion
  - $\Rightarrow$ scope restriction
  - $\Rightarrow$ method lookup

# *Denotational Semantics*

- clean theory
- specified from the denotations of the basic operations of the language
  - $\Rightarrow$ assignment
  - $\Rightarrow$ scope expansion
  - $\Rightarrow$ scope restriction
  - $\Rightarrow$ method lookup
  - $\Rightarrow$ method invocation, . . .

# *Denotational Semantics*



- clean theory
- specified from the denotations of the basic operations of the language
  $\Rightarrow$ assignment
  $\Rightarrow$ scope expansion
  $\Rightarrow$ scope restriction
  $\Rightarrow$ method lookup
  $\Rightarrow$ method invocation, ...
- compositional: $[\![c_1; c_2]\!] = [\![c_1]\!] \circ [\![c_2]\!]$

# *Denotational Semantics*

- clean theory
- specified from the denotations of the basic operations of the language
  $\Rightarrow$ assignment
  $\Rightarrow$ scope expansion
  $\Rightarrow$ scope restriction
  $\Rightarrow$ method lookup
  $\Rightarrow$ method invocation, ...
- compositional: $[\![c_1; c_2]\!] = [\![c_1]\!] \circ [\![c_2]\!]$
- easy to implement (no memoization)

# *Concrete and abstract $[\![c]\!]$ ????*

- it's an input/output map!

# *Concrete and abstract $[\![c]\!]$ ????*

- it's an input/output map!
- but it can be much richer [watchpoint semantics, Spoto, SAS 2001]

# *Concrete and abstract $[\![c]\!]$ ????*

- it's an input/output map!
- but it can be much richer [watchpoint semantics, Spoto, SAS 2001]
- it can be concrete or abstract

# *Concrete and abstract $[\![c]\!]$ ????*

- it's an input/output map!
- but it can be much richer [watchpoint semantics, Spoto, SAS 2001]
- it can be concrete or abstract
- if the abstract domain is finite, also the abstract map $[\![c]\!]$ is finite for every command $c$.

# *An Example: sign analysis*

Consider a program point with variables specified by a type environment $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

# *An Example: sign analysis*

Consider a program point with variables specified by a type environment $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

The command $\mathtt{a} := \mathtt{b}$ is denoted by $d = [\![\mathtt{a} := \mathtt{b}]\!]$ *i.e.,*

# *An Example: sign analysis*

Consider a program point with variables specified by a type environment $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

The command $\mathtt{a} := \mathtt{b}$ is denoted by $d = [\![\mathtt{a} := \mathtt{b}]\!]$ *i.e.,*

$$d(\mathtt{empty}) = \mathtt{empty}$$

$$d([+,+,+,+]) = [+,+,+,+] \quad d([+,+,+,-]) = [+,+,+,-]$$

$$d([+,+,-,+]) = [+,+,-,+] \quad d([+,+,-,\mathtt{u}]) = [+,+,-,\mathtt{u}]$$

$$d([+,-,+,+]) = [-,-,+,+] \quad d([+,-,\mathtt{u},-]) = [-,-,\mathtt{u},-]$$

$$d([+,-,-,+]) = [-,-,-,+] \qquad \ldots$$

# *Playing with Denotations*

- sequential composition $\circ$: it is functional composition

# *Playing with Denotations*

- sequential composition $\circ$: it is functional composition
  ! but a more precise composition can be used instead

# *Playing with Denotations*

- sequential composition $\circ$: it is functional composition
  ! but a more precise composition can be used instead
- disjunctive composition $\cup$: it is the pointwise lub over the abstract domain

# *Playing with Denotations*

- sequential composition $\circ$: it is functional composition
  ! but a more precise composition can be used instead
- disjunctive composition $\cup$: it is the pointwise lub over the abstract domain
  ? we use it for conditionals and virtual method calls

# *Playing with Denotations*

- sequential composition $\circ$: it is functional composition
  ! but a more precise composition can be used instead
- disjunctive composition $\cup$: it is the pointwise lub over the abstract domain
  ? we use it for conditionals and virtual method calls
- fixpoint: it is the limit of an ascending chain

# *Playing with Denotations*

- sequential composition $\circ$: it is functional composition
  ! but a more precise composition can be used instead
- disjunctive composition $\cup$: it is the pointwise lub over the abstract domain
  ? we use it for conditionals and virtual method calls
- fixpoint: it is the limit of an ascending chain
  ? we use it to handle recursion

# *Playing with Denotations*

- sequential composition $\circ$: it is functional composition
  ! but a more precise composition can be used instead
- disjunctive composition $\cup$: it is the pointwise lub over the abstract domain
  ? we use it for conditionals and virtual method calls
- fixpoint: it is the limit of an ascending chain
  ? we use it to handle recursion
  $\Rightarrow$ we need a check for function equality to stop the computation.

# *Is it a dream?*



The (apparent) dimension of $[\![c]\!]$ is $O(2^n)$ where $n$ is the number of integer variables in scope.

# *Is it a dream?*

The (apparent) dimension of $[\![c]\!]$ is $O(2^n)$ where $n$ is the number of integer variables in scope.

$\Downarrow$

Computing the abstract denotational semantics of a possible very small program looks computationally expensive.

# *Small Improvements*

$$d([+,+,+,+]) = [+,+,+,+]$$
$$d([+,+,+,-]) = [+,+,+,-]$$

entails that

$$d([+,+,+,\mathtt{u}]) =$$

# *Small Improvements*

$$d([+,+,+,+]) = [+,+,+,+]$$
$$d([+,+,+,-]) = [+,+,+,-]$$

entails that

$$d([+,+,+,\mathtt{u}]) = [+,+,+,\mathtt{u}]$$

# *The BDD Revolution*

R. E. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE ToC, 1986.

# *The BDD Revolution*

R. E. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE ToC, 1986.

It defines the binary decision diagrams (BDD's) to represent functions.

# *The BDD Revolution*

R. E. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE ToC, 1986.

It defines the binary decision diagrams (BDD's) to represent functions.

Very fast and compact in practice.

# *The BDD Revolution*

R. E. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE ToC, 1986.

It defines the binary decision diagrams (BDD's) to represent functions.

Very fast and compact in practice.

Why? Because many functions contain regularity or default cases that get exploited by the BDD's normal form.

# *How?*

# *How?*

1. Every map can be encoded into a Boolean function

# *How?*

1. Every map can be encoded into a Boolean function
2. Every Boolean function can be encoded into a graph

# *How?*

1. Every map can be encoded into a Boolean function
2. Every Boolean function can be encoded into a graph
3. Graphs can be kept in normal (minimal) form

# *How?*

1. Every map can be encoded into a Boolean function
2. Every Boolean function can be encoded into a graph
3. Graphs can be kept in normal (minimal) form
4. Operations over graphs correspond to $\circ$, $\cup$ and functional equality.

# *The Price to Pay*

- A large number of BDD nodes must be allocated to reduce garbage collection as much as possible

# *The Price to Pay*

- A large number of BDD nodes must be allocated to reduce garbage collection as much as possible

  $\Rightarrow$ a huge amount of memory is needed even for small programs

# *The Price to Pay*

- A large number of BDD nodes must be allocated to reduce garbage collection as much as possible

  $\Rightarrow$ a huge amount of memory is needed even for small programs

- Encodings complicate the implementation

# *The Price to Pay*

- A large number of BDD nodes must be allocated to reduce garbage collection as much as possible

  $\Rightarrow$ a huge amount of memory is needed even for small programs

- Encodings complicate the implementation

  $\Rightarrow$ we must find the encoding and program the encoder and the decoder (to show the output to the user).

# *Logic Variables*



We can use logic variables for a compact representation:

# *Logic Variables*

We can use logic variables for a compact representation:

Let again $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

# *Logic Variables*

We can use logic variables for a compact representation:

Let again $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

The command $\mathtt{a} := \mathtt{b}$ is denoted by $d = [\![\mathtt{a} := \mathtt{b}]\!]$ where

# *Logic Variables*

We can use logic variables for a compact representation:

Let again $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

The command $\mathtt{a} := \mathtt{b}$ is denoted by $d = [\![\mathtt{a} := \mathtt{b}]\!]$ where

$$d(\mathtt{empty}) = \mathtt{empty}$$

# *Logic Variables*

We can use logic variables for a compact representation:

Let again $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

The command $\mathtt{a} := \mathtt{b}$ is denoted by $d = [\![\mathtt{a} := \mathtt{b}]\!]$ where

$$d(\mathtt{empty}) = \mathtt{empty}$$
$$d([\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{O}]) = [\mathtt{B}, \mathtt{B}, \mathtt{C}, \mathtt{O}]$$

for all $A, B, C, O \in \{+, -\}$.

# *Logic Variables*

We can use logic variables for a compact representation:

Let again $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

The command $\mathtt{a} := \mathtt{b}$ is denoted by $d = [\![\mathtt{a} := \mathtt{b}]\!]$ where

$$d(\mathtt{empty}) = \mathtt{empty}$$
$$d([\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{O}]) = [\mathtt{B}, \mathtt{B}, \mathtt{C}, \mathtt{O}]$$

for all $A, B, C, O \in \{+, -\}$.

O(n) !!

# *Logic Programs*

The compact representation

$$d(\texttt{empty}) = \texttt{empty}$$

$$d([\texttt{A}, \texttt{B}, \texttt{C}, \texttt{O}]) = [\texttt{B}, \texttt{B}, \texttt{C}, \texttt{O}]$$

# *Logic Programs*

The compact representation

$$d(\texttt{empty}) = \texttt{empty}$$
$$d([\texttt{A},\texttt{B},\texttt{C},\texttt{O}]) = [\texttt{B},\texttt{B},\texttt{C},\texttt{O}]$$

is isomorphic to the logic program

```
io(empty,empty).
io([A,B,C,O],[B,B,C,O]).
```

# *Logic Programs*

The compact representation

$$d(\texttt{empty}) = \texttt{empty}$$
$$d([\texttt{A},\texttt{B},\texttt{C},\texttt{O}]) = [\texttt{B},\texttt{B},\texttt{C},\texttt{O}]$$

is isomorphic to the logic program

```
io(empty,empty).
io([A,B,C,O],[B,B,C,O]).
```

Only facts are used.

# *Another Example*

Let again $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

# *Another Example*

Let again $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

The command $\mathtt{a} := \mathtt{b} - \mathtt{1}$ is denoted by $d = [\![\mathtt{a} := \mathtt{b} - \mathtt{1}]\!]$ where

# *Another Example*

Let again $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

The command $\mathtt{a} := \mathtt{b} - \mathtt{1}$ is denoted by $d = [\![\mathtt{a} := \mathtt{b} - \mathtt{1}]\!]$ where

$$d(\mathtt{empty}) = \mathtt{empty}$$

# *Another Example*

Let again $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

The command $\mathtt{a} := \mathtt{b} - \mathtt{1}$ is denoted by $d = [\![\mathtt{a} := \mathtt{b} - \mathtt{1}]\!]$ where

$$d(\mathtt{empty}) = \mathtt{empty}$$

$$d([\mathtt{A}, -, \mathtt{C}, \mathtt{O}]) = [-, -, \mathtt{C}, \mathtt{O}]$$

# *Another Example*

Let again $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

The command $\mathtt{a} := \mathtt{b} - \mathtt{1}$ is denoted by $d = [\![\mathtt{a} := \mathtt{b} - \mathtt{1}]\!]$ where

$$d(\mathtt{empty}) = \mathtt{empty}$$
$$d([\mathtt{A}, -, \mathtt{C}, \mathtt{O}]) = [-, -, \mathtt{C}, \mathtt{O}]$$
$$d([\mathtt{A}, +, \mathtt{C}, \mathtt{O}]) = [\mathtt{u}, +, \mathtt{C}, \mathtt{O}]$$

for all $A, C, O \in \{+, -\}$.

# *Another Example*

Let again $\tau = [\mathtt{a} \mapsto int, \mathtt{b} \mapsto int, \mathtt{c} \mapsto int, \mathtt{out} \mapsto int]$

The command $\mathtt{a} := \mathtt{b} - \mathtt{1}$ is denoted by $d = [\![\mathtt{a} := \mathtt{b} - \mathtt{1}]\!]$ where

$$d(\mathtt{empty}) = \mathtt{empty}$$
$$d([\mathtt{A}, -, \mathtt{C}, \mathtt{O}]) = [-, -, \mathtt{C}, \mathtt{O}]$$
$$d([\mathtt{A}, +, \mathtt{C}, \mathtt{O}]) = [\mathtt{u}, +, \mathtt{C}, \mathtt{O}]$$

for all $A, C, O \in \{+, -\}$.

still O(n) !!

# *Similar Techniques*

- Type analysis of ML
  [Hindley, TAMS, 1969]
  [Milner, JCSS, 1978]

# *Similar Techniques*

- Type analysis of ML
  [Hindley, TAMS, 1969]
  [Milner, JCSS, 1978]
- Type Analysis of logic programs
  [Codish & Demoen, SAS, 1994]
  [Howe & King, ESOP, 2000]
  [Levi & Spoto, PDP, 1998]

# *Similar Techniques*

- Type analysis of ML
  [Hindley, TAMS, 1969]
  [Milner, JCSS, 1978]
- Type Analysis of logic programs
  [Codish & Demoen, SAS, 1994]
  [Howe & King, ESOP, 2000]
  [Levi & Spoto, PDP, 1998]
- Mode Analysis of logic programs through pre-interpretations
  [Gallagher & al., ILPS, 1995]

# *Similar Techniques*

- Type analysis of ML
  [Hindley, TAMS, 1969]
  [Milner, JCSS, 1978]
- Type Analysis of logic programs
  [Codish & Demoen, SAS, 1994]
  [Howe & King, ESOP, 2000]
  [Levi & Spoto, PDP, 1998]
- Mode Analysis of logic programs through pre-interpretations
  [Gallagher & al., ILPS, 1995]

No general theory.

# *Similar Techniques*

- Type analysis of ML
  [Hindley, TAMS, 1969]
  [Milner, JCSS, 1978]
- Type Analysis of logic programs
  [Codish & Demoen, SAS, 1994]
  [Howe & King, ESOP, 2000]
  [Levi & Spoto, PDP, 1998]
- Mode Analysis of logic programs through pre-interpretations
  [Gallagher & al., ILPS, 1995]

No general theory. No application to imperative programs

# *Compact Denotations*

A compact denotation is a possibly non-ground set of arrows

$$cd = \{L_1 \to r_1, \ldots, L_m \to r_m\}$$

$$cd(L_1) = r_1$$

$$\vdots$$

$$cd(L_m) = r_m$$

# *Compact Denotations*

A compact denotation is a possibly non-ground set of arrows

$$cd = \{L_1 \to r_1, \ldots, L_m \to r_m\}$$

$$cd(L_1) = r_1$$

$$\vdots$$

$$cd(L_m) = r_m$$

Its meaning is the denotation $\overline{cd}$ obtained by grounding the variables *w.r.t.* a set of legal substitutions.

# *Constraints on Compact Denotations*



We require that compact denotations:

# *Constraints on Compact Denotations*

We require that compact denotations:

- have exhaustive heads

# *Constraints on Compact Denotations*



We require that compact denotations:

- have exhaustive heads
- have non overlapping heads

# *Constraints on Compact Denotations*



We require that compact denotations:

- have exhaustive heads
- have non overlapping heads
- have monotonic meaning.

# *Composition of Compact Denotations*

How can we mimic $\circ$ with an operation $\circ^*$ over compact denotations?

# Composition of Compact Denotations

How can we mimic $\circ$ with an operation $\circ^*$ over compact denotations?

$$t_1 \circ^* t_2 = \left\{ (l_2 \to r_1)\theta \;\middle|\; \begin{array}{l} l_2 \to r_2 \in t_2,\ l_1 \to r_1 \in t_1 \\ \texttt{domain\_entails}(l_1, r_2) \\ \text{computes } \theta \end{array} \right\}.$$

# Composition of Compact Denotations

How can we mimic $\circ$ with an operation $\circ^*$ over compact denotations?

$$t_1 \circ^* t_2 = \left\{ (l_2 \to r_1)\theta \;\middle|\; \begin{array}{l} l_2 \to r_2 \in t_2,\ l_1 \to r_1 \in t_1 \\ \texttt{domain\_entails}(l_1, r_2) \\ \text{computes } \theta \end{array} \right\}.$$

`domain_entails` is domain-dependent!

# An Example

$$\texttt{empty} \rightarrow \texttt{empty}$$
$$[\texttt{B}, +] \rightarrow [-]$$
$$[\texttt{B}, -] \rightarrow [+]$$

# An Example

$$\begin{array}{lcl} \texttt{empty} \to \texttt{empty} & & \texttt{empty} \to \texttt{empty} \\ [\texttt{B}, +] \to [-] & \circ^* & [\texttt{A}, +] \to [-, \texttt{u}] \\ [\texttt{B}, -] \to [+] & & [\texttt{A}, -] \to [+, +] \end{array}$$

$$=$$

# An Example

$$\begin{array}{lcl} \texttt{empty} \to \texttt{empty} & & \texttt{empty} \to \texttt{empty} \\ [\mathtt{B}, +] \to [-] & \circ^* & [\mathtt{A}, +] \to [-, \mathtt{u}] \\ [\mathtt{B}, -] \to [+] & & [\mathtt{A}, -] \to [+, +] \end{array}$$

$$\texttt{empty} \to \texttt{empty}$$

$$=$$

# An Example

$$\begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{B}, +] \to [-] \\ [\texttt{B}, -] \to [+] \end{array} \quad \circ^* \quad \begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{A}, +] \to [-, \texttt{u}] \\ [\texttt{A}, -] \to [+, +] \end{array}$$

$$= \begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{A}, +] \to [-] \end{array}$$

# An Example

$$\begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{B}, +] \to [-] \\ [\texttt{B}, -] \to [+] \end{array} \quad \circ^* \quad \begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{A}, +] \to [-, \texttt{u}] \\ [\texttt{A}, -] \to [+, +] \end{array}$$

$$= \begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{A}, +] \to [-] \\ [\texttt{A}, +] \to [+] \end{array}$$

# An Example

$$\begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{B}, +] \to [-] \\ [\texttt{B}, -] \to [+] \end{array} \quad \circ^* \quad \begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{A}, +] \to [-, \texttt{u}] \\ [\texttt{A}, -] \to [+, +] \end{array}$$

$$= \begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{A}, +] \to [-] \\ [\texttt{A}, +] \to [+] \\ [\texttt{A}, -] \to [-] \end{array}$$

# An Example

$$\begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{B}, +] \to [-] \\ [\texttt{B}, -] \to [+] \end{array} \quad \circ^* \quad \begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{A}, +] \to [-, \texttt{u}] \\ [\texttt{A}, -] \to [+, +] \end{array}$$

$$= \begin{array}{l} \texttt{empty} \to \texttt{empty} \\ [\texttt{A}, +] \to [-] \\ [\texttt{A}, +] \to [+] \\ [\texttt{A}, -] \to [-] \end{array} \qquad ([\texttt{A}, +] \to [\texttt{u}] \;!!)$$

This is not a compact denotation!

# *The Rest of the Story...*

- The result of $\circ^*$ must be normalised by factoring overlapping heads

# *The Rest of the Story...*

- The result of $\circ^*$ must be normalised by factoring overlapping heads
- We have proved that the resulting operation exactly mimics $\circ$ over compact denotations

# *The Rest of the Story...*

- The result of $\circ^*$ must be normalised by factoring overlapping heads
- We have proved that the resulting operation exactly mimics $\circ$ over compact denotations
- $\cup$ can be implemented in terms of $\circ^*$ and the lub of the abstract domain [Spoto, SAS 2001]

# *The Rest of the Story...*

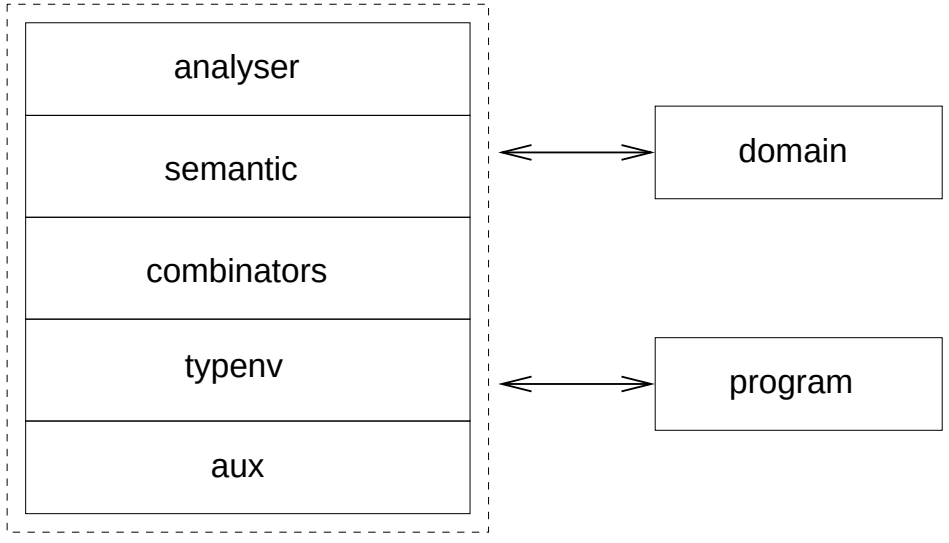
- The result of $\circ^*$ must be normalised by factoring overlapping heads
- We have proved that the resulting operation exactly mimics $\circ$ over compact denotations
- $\cup$ can be implemented in terms of $\circ^*$ and the lub of the abstract domain [Spoto, SAS 2001]
- for every basic operation of the language we have provided basic compact denotations

# *The Rest of the Story...*

- The result of $\circ^*$ must be normalised by factoring overlapping heads
- We have proved that the resulting operation exactly mimics $\circ$ over compact denotations
- $\cup$ can be implemented in terms of $\circ^*$ and the lub of the abstract domain [Spoto, SAS 2001]
- for every basic operation of the language we have provided basic compact denotations
- equality on compact denotations is first checked syntactically and then semantically.

# *LOOP*

The Localised Analyser for Object-Oriented Programs...




www.sci.univr.it/∼spoto/loop

# *Our Abstract Domains*

| analysis | ground | non-ground | BDD's |
|---|---|---|---|
| sign | `signs` | `signs_vars` | `signs_bdd` |
| rapid type | `rt` | `rt_vars` | |
| dataflow class | `df` | `df_vars` | |
| P&S class | `ps` | `ps_vars` | |
| escape | `e` | `e_vars` | |

# *Sign Analysis*

| Bmrk | signs | | signs_vars | |
|---|---|---|---|---|
| | time | space | time | space |
| arith | 108.578 | 13617292 | 0.163 | 26242 |

# *Rapid Type Analysis*

| Bmrk | rt | | rt_vars | |
|---|---|---|---|---|
| | time | space | time | space |
| figures | 2.956 | 776338 | 1.193 | 295880 |

# *Dataflow Class Analysis*

| Bmrk | df | | df_vars | |
|---|---|---|---|---|
| | time | space | time | space |
| clone | 11.450 | 2320420 | 0.078 | 18935 |

# *P&S Class Analysis*

| Bmrk | ps | | ps_vars | |
|---|---|---|---|---|
| | time | space | time | space |
| clone | 284.624 | 29393040 | 0.165 | 47855 |

# *Escape Analysis*

| Bmrk | e | | e_vars | |
|---|---|---|---|---|
| | time | space | time | space |
| inv | 0.432 | 14060 | 0.217 | 9268 |

# *Comparison with BDD's*

| Bmrk | signs_vars | | signs_bdd *small* | |
|---|---|---|---|---|
| | time | space | time | space |
| arith | 0.198 | 38485 | 0.481 | 1000 |

| Bmrk | signs_vars | | signs_bdd *large* | |
|---|---|---|---|---|
| | time | space | time | space |
| arith | 0.198 | 38485 | 0.168 | 1000000 |

# *Discussion*

- Logic programs as compact denotations are much faster and cheaper than ground denotations

# *Discussion*

- Logic programs as compact denotations are much faster and cheaper than ground denotations
- Their efficiency is similar to that of a BDD implementation. . .

# *Discussion*

- Logic programs as compact denotations are much faster and cheaper than ground denotations
- Their efficiency is similar to that of a BDD implementation. . .
- . . . which is more complicated to write!

# *Discussion*

- Logic programs as compact denotations are much faster and cheaper than ground denotations
- Their efficiency is similar to that of a BDD implementation. . .
- . . . which is more complicated to write!
- . . . and requires many statically allocated BDD nodes

# *Discussion*

- Logic programs as compact denotations are much faster and cheaper than ground denotations
- Their efficiency is similar to that of a BDD implementation. . .
- . . . which is more complicated to write!
- . . . and requires many statically allocated BDD nodes
- They are less effective for the domains which do not use a per-variable approximation (rapid type and escape analyses).

# *The Future*



- Implementation for the Java Virtual Machine

# *The Future*

- Implementation for the Java Virtual Machine
- Application to the verification of JML's `assignable` specifications [Spoto & Poll, FOOL-10, 2003]

# *The Future*

- Implementation for the Java Virtual Machine
- Application to the verification of JML's `assignable` specifications [Spoto & Poll, FOOL-10, 2003]
- Application to on-card verification?

# Q & A