The Complexity of Object Oriented Type Analysis

Yossi Gil

Computer Science, Technion

and IBM T.J. Watson Research Center

joint work with Alon Itai

ECOOP'98, Brussels Belgium, July 24, 1998

The Problem

- **Given:** an OO (class hierarchy) program
- **Find:** the set of dynamic types that each polymorphic variable may contain at any given program point
- **Example:**

```
struct B { virtual void f(); } b, *p;
struct D1: B { void f(); } d1;
struct D2: B { void f(); } d2;

void B::f()  { p = d1; p->f(); }
void D1::f() { p = d2; p->f(); }
void D2::f() { p = d1; }

int main()
{
    p = random() > 0.5 ? d1 : b;
    p->f();

    // What is the dynamic type(s) of p here?
}
```

Motivation

- **Polymorphic calls are expensive:**
  - **Dynamic dispatch mechanism:** determine the dynamic type of an object, then call the right method
  - **Lost optimization opportunities:** difficult to inline call
    - Many methods are very small
- **Polymorphic calls are rare:**
  - **A large portion of call sites are monomorphic**
  - **A large portion of actual calls are monomorphic**
- **Static analysis (state of the art):**
  - **Complex algorithms**
  - **Time consuming**
  - **Inaccurate**
  - **Heuristics, difficult to compare**

An Undecidable Problem?

```
void g()
{
    if (main() == 0)
        p = new D1;
    else
        p = new D2;

    // What is the dynamic type(s) of p here?
}
```

The fundamental hypothesis of static analysis

All paths in the control-flow graph are probable

Dimensions of Static Analysis

- **Control flow:**
  - **Flow insensitive:** ignore control flow information. Statements may be executed in an arbitrary order and arbitrarily many times
    - Used e.g., in standard and non-standard type inference
  - **Flow sensitive:** analysis may not infer the truth value of conditions (e.g., in **if** and **loop** statements)
  - **Object Oriented:** analysis may deduce (and use) the actual method called
  - **Heuristic:** analysis infer (and use) the truth vale of conditions
- **Data types:** simple / multi-level pointer/ recursive
- **Program structure:** intra-procedural / non-recursive / recursive

Challenge: determine the complexity of every combination of the parameters

Multi-Level Pointers

```
template <unsigned n>
    class S {
        public:
            S<n-1> *left, *right;
    };

#define N 1000

class S<N> a;
```

Reminder: Complexity Classes

- **Time complexity classes:**
  - **P, NP, EXP-TIME, DEXP-TIME, ...**
- **Space complexity classes:**
  - **P-SPACE, EXP-SPACE, DEXP-SPACE, …**
- **Relationship:**

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{P\text{-}SPACE} \subseteq \mathrm{EXP\text{-}TIME} \subset \mathrm{EXP\text{-}SPACE} \subseteq \mathrm{EXP\text{-}TIME}$$

- **Problem P is Complexity Class *<u>X-hard</u>*:**
  - **P is more difficult than <u>all</u> problems in X.**
- **Problem P is Complexity Class *<u>X-complete</u>*:**
  - **P is X-hard**
  - **P is in X**

Previous Results

- **Pande and Ryder: type analysis is NP-Hard**
  - **object oriented analysis**
  - **non-recursive data structures**
  - **non-recursive programs**
    - (intra-procedural is not interesting in OO analysis)
- **Hurwitz: alias-analysis (may point-to) is NP-hard**
  - **flow insensitive**
  - **recursive data-structures**
  - **non-recursive programs**

Our Results

Recursion	Multi-level pointers	Recursive Data Types	Complexity
No	No	No	P-SPACE complete
Yes	No	No	EXP-TIME complete
No	Yes	No	EXP-SPACE complete
Yes	Yes	No	DEXP-TIME complete
—	—	Yes	Undecidable

P-Space Hard

- **Basic idea**: reduce the halting problem of a linear bounded automaton.
- **LBA:** a Turing machine whose tape length is bounded by a linear function of the input size.
- **Ingredients of a Turing machine:**
  - **Q:** a finite set of internal states
  - $\mathbf{q_0} \in$ **Q:** an initial state
  - $\mathbf{f} \in$ **Q;** an accepting state
  - **T:** a working tape of length n of {0,1} symbols
  - δ: $\mathbf{Q} \times \{0,1\} \rightarrow \mathbf{Q} \times \{0,1\} \times \{\text{prev, next}\}$: a transition function, specifying for each internal state and current tape head value:
    - next state
    - value to write on the tape
    - direction of head movement

Encoding Techniques

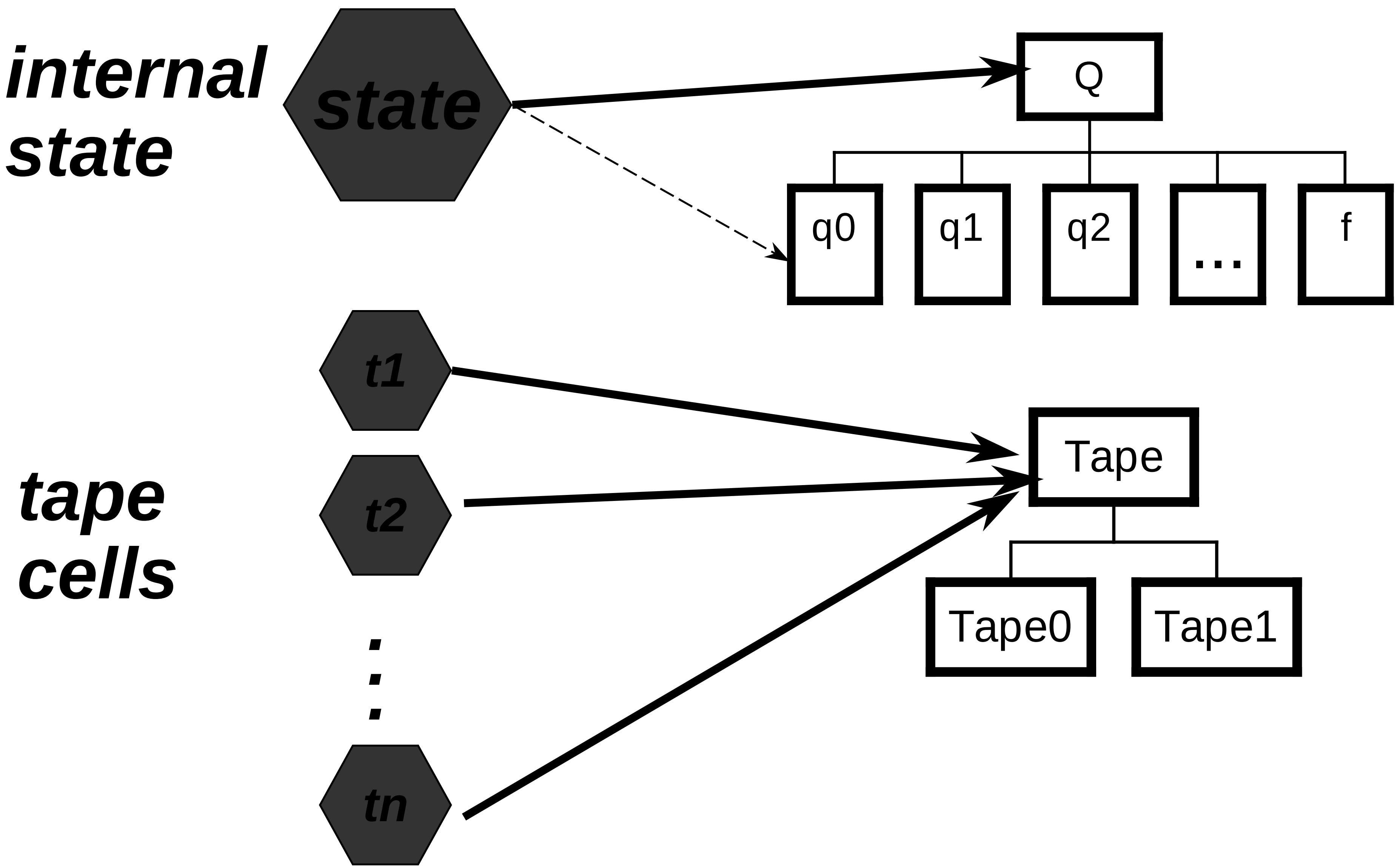
- **Type encoding:**
  - **Value:** Each distinct value in a Turing machine is encoded as a distinct type.
  - **Variable:** A variable of a Turing machine which may assume different values, is encoded as a polymorphic program variable which may assume different dynamic types
  - **Examples:**
    - Set of internal states / current LBA state
    - {0,1} set of tape cell values / each tape cell
- **Dynamic binding encoding:** a different implementation of a virtual function for each different type representing value
- **Multiple dispatch:** to realize a function depending on several values
- **Main difficulty:** encode tape head position

Encoding LBA’s Current State

```
class Q {
    public:
        virtual void delta0() = 0;
        virtual void delta1() = 0;
};
```

```
class Tape {
    public:
        virtual void delta() = 0;
};
class Tape0: public Tape {
    public:
        virtual void delta() { state->delta0{} }
};
class Tape1: public Tape {
    public:
        virtual void delta() { state->delta1(); }
};
```

Class Object Diagram



Initialization and Termination

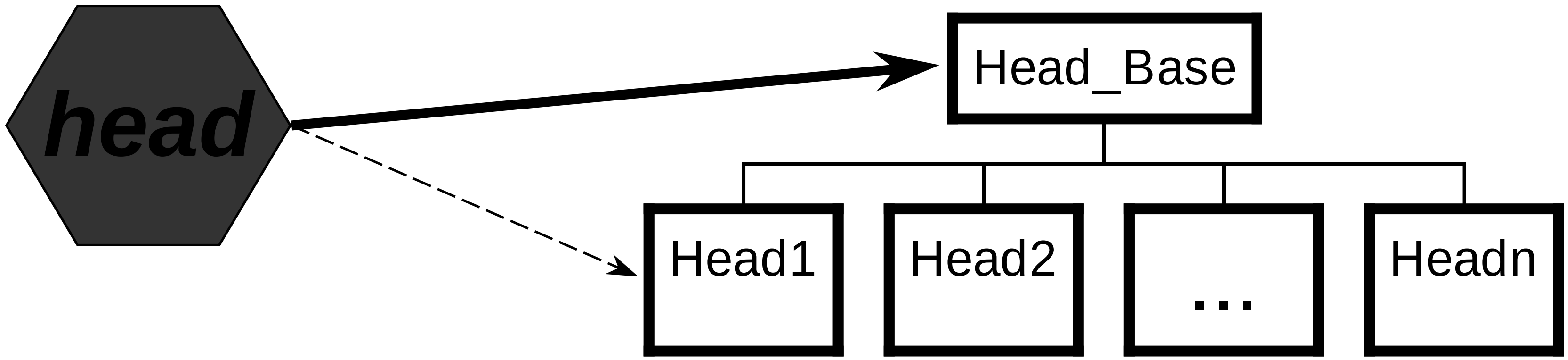
```
Q *state = new Q0;

Tape *t1 = new Tape0;
Tape *t2 = new Tape1;
…
Tape *tn = new Tape0;

main()
{
    … // Turing machine emulation goes here

    // Is F a dynamic type of state?
}
```

Encoding Tape Head Position




```
class Head_base {
   public:
      virtual void next() = 0;
      virtual void prev() = 0;
      virtual void set0() = 0;
      virtual void set1() = 0;
      virtual void delta() = 0;
} *head;
```

Using a Template for Generating Classes for Head Positioning

```
template <int i>
class Head: public Head_base {
public:
    static Tape *t;
    void next() { head = new Head<i+1>; }
    void prev() { head = new Head<i-1>; }
    void set0() {
      Head<i>::t = new Tape0;
    }
    void set1() {
      Head<i>::t = new Tape1;
    }
    void delta() {
      Head<i>::t -> delta();
    }
};
```

Template specialization for boundary cases not shown

Putting it all Together: the Delta function

- **Consider the following polymorphic call**

```
head->delta()
```

- **Suppose that** `head` **is of type** `Head<17>` (head is on position 17 on tape)

```
void Head<17>::delta() {
   Head<17>::t->delta(); }
```

- **Suppose that** `Head<17>::t` **is of type** `Tape1` (cell 17 on the tape has 1)

```
void Tape1::delta() {
   Head<17>::state->delta1(); }
```

- **Suppose that** `state` **is of type** `Q4` (machine is in internal state 4)

...

The Delta Function (cont.)

- **Suppose that** `state` **is of type** `Q4` (machine is in internal state 4)

```
void Q4::delta0() {
  state = new Q7; // State transition
  head->set0();
  head->next();
}
```

- **Writing to tape:**

```
void Head<17>::set0() {
  Head<17>::t = new Tape0;
}
```

- **Moving the head**

```
void Head<17>::next() {
  head = new Head<18>;
}
```

The Main Function

```
main()
{
    head = new Head<0>;

    for (;;) {
      head -> delta();
    }
    //  What is the type of state here?
}
```

- **No recursion used**
- **No recursive data structures**
- **As flow-insensitive as OO analysis can be**
- **O(n) classes**
- **O(1) Inheritance depth**

P-Space Completeness

- **Assume:**
  - **No recursive data types**
  - **No recursive calls**
  - **All variables are global**
- **Reduction to:** s-t connetivity in exponentially sized graph
  - **Nodes:** a tuple of (1) program counter location, (2) function call stack, and (3) an assignment of types to all program variables
    - Each node is described by a vector of polynomial size
    - Total number of nodes is exponential
  - **Statements:**
    - **Simple:** a simple edge in the graph
    - **If statement:** two edges, each corresponding to a branch
    - **Loop condition:** same as “if” statement
    - **Loop body:** a loop edge
    - **Function call:** an edge

Dealing with Recursion

- **Main Difficulty:** memory is unbounded...
- **Auxiliary push-down machine (aux-PDM):** Turing machine + Unbounded stack.
- **Theorem (Cook)** ***:*** Let *w(n)* >= *lg(n)*. Let *A* be a language. Then, the following are equivalent:
  - **A is accepted by some <u>deterministic</u> aux-PDM within storage *w(n)*.**
  - **A is accepted by some <u>non-deterministic</u> aux-PDM within storage *w(n)*.**
  - **A is accepted by some time-bounded computer within time $2^{c\, w(n)}$, for some constant c > 0.**

Other Complexity Results

- **EXP-TIME:** ***Reduction to an aux-PDA (Turing machine with stack)***
- **EXP-SPACE:** *With multi-level pointers, space might be exponential*
  - **simulate a turing machine with exponential size tape.**
- **DEXP-TIME:** Same as EXP-TIME, but with exponential sized tape.
- **UNDECIDABLE:** Space is unbounded, hence can simulate a general turing machine.

Scope Size

- **Scope size of a routine:** *total number of accessible variables*
  - **Parameters**
  - **Local variables**
  - **Global variables**
  - **Instance variables (defined and inherited)**
- **Scope size of a program:** ***maximal SS over all routines***
- ***Theorem:*** if the SS of a program p of size n is at most log n, then, OOTA of p can be done in sub-exponential time: $n^{O(\log(n))}$
  - **Recursion allowed**
  - **No recursive data types**
  - **Multi-level pointers to the extent allowed by SS <= log n**

Chameleonicity

- **A class C:** *Chm(C) := Rnd(lg(N) + 0.5).*
  - **N= Number of concrete classes that inherit from C, including C itself.**
  - **Chm(C)= # bits required to represent the run time type a variable of type C.**
- **A varaible V of type C:** *Chm(V) := Chm(V)*
- **A routine R:** *Chm(R) := Sum(Chm(V), V in R)*
  - **Chm(p) = # bits required to represent the run time type of all variables in R**
- **A prograp P:** Chm(P) := Max(Chm(R), R in P)
  - **Hypothesis:** ***for "well written programs", chm(P) = O(lg n)***
- **Theorem:** the time for solving OOTA of P is

$$2^{c*chm(P)}$$

Open Problems

- Exact characterization of the may-alias, flow-insensitive analysis problem
- Study of the flow-insensitive model of computation.

- Barth J.
- CACM vol 21 # 9 , page 724-736
- A practical inter procedural data flow analysis

Other Results

- Basic: No-recursion/Simple data types
  - P-Space Complete
- + Recursion:
  - EXP-TIME complete
- Basic + Multi-level pointers
  - EXP-SPACE complete
- Basic + Multi-level pointers + recursion
  - DOUBLE-EXP-TIME COMPLETE
- Recursive data structures
  - Undecidable!
- No global variables, and each function has only a constant number of local variables, no arrays. Each type has a fixed number of slots?
  - Equivalence of two programs is decidable.
  - Langmaack, 72/3 acta informatica.
  - Passing parametes to programs.
  - If each type is finite then the halting program is decidable.

G

- **Golubski, Wolfgang, PHD: Smalltalk programs are decidable method is formal reachable.**
  - **Control-flow sensitive**
  - **Inter-procedural analysis**
- **Reference.**