Resource Control

David Aspinall

School of Informatics, University of Edinburgh

Work of Mobius participants from
UPM/UCM Madrid, IRISA/INRIA Rennes, LMU Munich, Univ. of Edinburgh.

Mobius Tutorial at ETAPS
Sunday, 1st April 2007

Outline

1. Resource control overview
2. Heap space analysis
3. Permission analysis
4. Cost Analysis
5. Conclusions

Varieties of resource

1. Machine resources — properties of execution
   - **heap space**
   - **execution time**
   - stack height, call counting, ...
2. Internal resources — manipulated in code
   - **use-once permissions**
   - collection sizes, thread pools, ...
3. External resources — exist outwith JVM
   - **billable events**
   - user interaction, power usage...

Controlling resources

- Main interest: *quantitative analysis* of resource usage.
  - Elsewhere: *patterns of access*, e.g. create, open, close.
- Aims:
  - Type-based (simple specification) paradigms for useful cases
  - Static analysis to predict behaviour
  - Certification for PCC
- Benefits of resource control:
  - Obvious **security** relevance. Most security breaches amount to violating resource control: exceeding allowed bounds or gaining unauthorised access to resources.
  - Also useful beyond security: feasibility, scheduling, pricing.

Analysis with type systems

Focus of this talk: two analyses based on *type systems*:

- Heap space
  - type system approach for separating property from inference
  - translation to JVM and Base Logic
  - Munich: Beringer, Hofmann
- Permissions, internally
  - Java API for *resource managers*, runtime checks
  - type system with simple logical constraints, erasibility of checks
  - Edinburgh: Maier, Stark, Aspinall

Analysis with control flow graphs

Other analyses using control flow graphs (classical program analysis):

- Permissions, externally
  - native methods given permissions profile
  - static analysis of control flow graph, check no errors
  - Rennes: Besson, Jensen, Pichardie.
  - see: WITS'07 invited talk by Thomas Jensen
- Execution cost
  - assign costs to bytecode statements (e.g., time)
  - generate set of cost equations on control flow graph
  - Madrid: Albert, Arenas, Genaim, Puebla, Zanardini.
  - see: ESOP'07 paper by Madrid team (brief recap at end)

Type systems approach

Inductively defined *typing judgement* relates program phrases (e) to types (τ) given an assignment of types (Γ) to methods and variables.

Typing rules are mostly *syntax-directed* $\frac{\Gamma \vdash x_1 : \mathtt{int} \quad \Gamma \vdash x_2 : \mathtt{int}}{\Gamma \vdash x_1 + x_2 \mathtt{int}}$ but ...

- side conditions involving constraints (numerical or set-based)
- types for methods must be given
- existential metavariables, e.g. in subsumption rule: $\frac{e:\tau \quad \tau \leq \tau'}{e:\tau'}$

Type soundness: valid typing implies desired semantic property.

Explaining power: simple formulation of declarative presentation explains type errors to user.

Generic inference algorithm: merge in non-syntax directed rules; gather constraints from existential metavariables; iterate to find typings for methods.

Type systems approach

Inductively defined *typing judgement* relates program phrases (e) to types (τ) given an assignment of types (Γ) to methods and variables.

Typing rules are mostly *syntax-directed* $\frac{\Gamma \vdash x_1 : \texttt{int} \quad \Gamma \vdash x_2 : \texttt{int}}{\Gamma \vdash x_1 + x_2 \texttt{int}}$ but ...

- side conditions involving constraints (numerical or set-based)
- types for methods must be given
- existential metavariables, e.g. in subsumption rule: $\frac{e:\tau \quad \tau \leq \tau'}{e:\tau'}$

Type soundness: valid typing implies desired semantic property.

Explaining power: simple formulation of declarative presentation explains type errors to user.

Generic inference algorithm: merge in non-syntax directed rules; gather constraints from existential metavariables; iterate to find typings for methods.

Type systems approach

Inductively defined *typing judgement* relates program phrases (e) to types (τ) given an assignment of types (Γ) to methods and variables.

Typing rules are mostly *syntax-directed* $\frac{\Gamma \vdash x_1 : \mathtt{int} \quad \Gamma \vdash x_2 : \mathtt{int}}{\Gamma \vdash x_1 + x_2 \mathtt{int}}$ but ...

- side conditions involving constraints (numerical or set-based)
- types for methods must be given
- existential metavariables, e.g. in subsumption rule: $\frac{e : \tau \quad \tau \leq \tau'}{e : \tau'}$

Type soundness: valid typing implies desired semantic property.

Explaining power: simple formulation of declarative presentation explains type errors to user.

Generic inference algorithm: merge in non-syntax directed rules; gather constraints from existential metavariables; iterate to find typings for methods.

Type systems approach

Inductively defined *typing judgement* relates program phrases (e) to types (τ) given an assignment of types (Γ) to methods and variables.

Typing rules are mostly *syntax-directed* $\frac{\Gamma \vdash x_1 : \texttt{int} \quad \Gamma \vdash x_2 : \texttt{int}}{\Gamma \vdash x_1 + x_2 \texttt{int}}$ but ...

- side conditions involving constraints (numerical or set-based)
- types for methods must be given
- existential metavariables, e.g. in subsumption rule: $\frac{e : \tau \quad \tau \leq \tau'}{e : \tau'}$

Type soundness: valid typing implies desired semantic property.

Explaining power: simple formulation of declarative presentation explains type errors to user.

Generic inference algorithm: merge in non-syntax directed rules; gather constraints from existential metavariables; iterate to find typings for methods.

Type systems approach

Inductively defined *typing judgement* relates program phrases (e) to types (τ) given an assignment of types (Γ) to methods and variables.

Typing rules are mostly *syntax-directed* $\frac{\Gamma \vdash x_1 : \mathtt{int} \quad \Gamma \vdash x_2 : \mathtt{int}}{\Gamma \vdash x_1 + x_2 \mathtt{int}}$ but ...

- side conditions involving constraints (numerical or set-based)
- types for methods must be given
- existential metavariables, e.g. in subsumption rule: $\frac{e:\tau \quad \tau \leq \tau'}{e:\tau'}$

Type soundness: valid typing implies desired semantic property.

Explaining power: simple formulation of declarative presentation explains type errors to user.

Generic inference algorithm: merge in non-syntax directed rules; gather constraints from existential metavariables; iterate to find typings for methods.

Comparison: Type system $\leftrightarrow$ Program Analysis

Advantages of type systems:

- Soundness and inference algorithms separated. No need to understand inference algorithm to grasp meaning of type system.
- In general "interprocedural".
- Interaction with user, e.g. via type annotations.
- Perhaps easier to interface with program logics.

Disadvantages of type systems:

- Less experience than with program analysis.
- More often ad-hoc due to lack of approximation theory.
- Sometimes typing rules become very complicated.

Outline

1 Resource control overview

2 Heap space analysis

3 Permission analysis

4 Cost Analysis

5 Conclusions

Simple type system for heap space

Types are natural numbers.
Given assignment of types to functions (methods), assign types to expressions with rules like:

$$\frac{\vdash e : n \qquad n \leq m}{e : m} \quad \text{(TWEAK)}$$

$$\frac{\vdash e_1 : n_1 \qquad \vdash e_2 : n_2}{\vdash \texttt{let}\ \ x\texttt{=}e_1\ \texttt{in}\ \ e_2 : n_1 + n_2} \quad \text{(TLET)}$$

$$\frac{}{\vdash \texttt{new}\ \ C(\mathbf{x}) : 1} \quad \text{(TNEW)}$$

$$\frac{\Sigma(\texttt{f}) = n}{\vdash \texttt{f}(x_1, \ldots, x_n) : n} \quad \text{(TAPP)}$$

Type inference

- Assign a variable to each function symbol
- Derive "skeleton" type derivation using TWEAK only next to TAPP.
- Try to solve resulting constraints.
- Provably equivalent to graph-based analysis.
- Can be extended to deallocation and input-dependent bounds.
- Result:
  - Static evidence that program satisfies some resource bound
  - Security model: client refuses to execute code that has no bound, or bound beyond device limits.

Extending to deallocation

Typing judgement: $\Gamma \vdash e : m \to n$.
Meaning: If freelist has size $s \geq m$ then evaluation of e will succeed and leave freelist of size $\geq n + s - m$.

$$\frac{\vdash e : m \longrightarrow n \qquad m' \geq m \qquad n' \leq n + m' - m}{e : m' \longrightarrow n'} \quad (\text{TWEAK})$$

$$\frac{\vdash e_1 : m \longrightarrow k \qquad \vdash e_2 : k \longrightarrow n}{\vdash \texttt{let}\ \ x\texttt{=}e_1\ \texttt{in}\ \ e_2 : m \longrightarrow n} \quad (\text{TLET})$$

$$\frac{}{\vdash \texttt{new}\ \ C(\mathbf{x}) : 1 \longrightarrow 0} \quad (\text{TNEW})$$

$$\frac{\Sigma(\texttt{f}) = m \longrightarrow n}{\vdash \texttt{f}(x_1, \ldots, x_n) : m \longrightarrow n} \quad (\text{TAPP})$$

Certification with type systems

- Typing derivations can be used directly as certificates. But: need to believe or understand type soundness.
- Likewise, successful runs of a program analysis, perhaps in condensed form can be used as certificates. But: need to believe or understand correctness of analysis.

Two better options:

- Formally prove correctness of analysis / type system
- Translate typing judgements into judgements of a formalised program logic, translate typing derivations into proofs of those translations.

Translating heap space type system into program logic

$e : n$ becomes $[\![e]\!] : \phi_n$ where ϕ_n is the specification

$$\phi_n \equiv \begin{array}{l} \texttt{pre} \\ \texttt{post}\,|h| \leq |h_{\mathrm{old}}| + n \\ \texttt{inv}\,|h| \leq |h_{\mathrm{old}}| + n \end{array}$$

Here $[\![e]\!]$ is the bytecode corresponding to e and $|h|$ is the size of the current heap, etc.

Typing derivations can then be translated into program logic rule by rule. Each typing rule can be translated (as an admissible rule) once and for all.

Mobius Base Logic expresses these assertions and provides proof rules for them that enable modular translation of typing rules. Resources other than heap size can be tracked with ghost fields in the heaps.
[See Base Logic lecture by Beringer for more].

Translating heap space type system into program logic

$e : n$ becomes $[\![e]\!] : \phi_n$ where ϕ_n is the specification

$$\phi_n \equiv \begin{array}{l} \texttt{pre} \\ \texttt{post}\,|h| \leq |h_{\mathrm{old}}| + n \\ \texttt{inv}\,|h| \leq |h_{\mathrm{old}}| + n \end{array}$$

Here $[\![e]\!]$ is the bytecode corresponding to e and $|h|$ is the size of the current heap, etc.

Typing derivations can then be translated into program logic rule by rule. Each typing rule can be translated (as an admissible rule) once and for all.

Mobius Base Logic expresses these assertions and provides proof rules for them that enable modular translation of typing rules. Resources other than heap size can be tracked with ghost fields in the heaps.
[See Base Logic lecture by Beringer for more].

Outline

1. Resource control overview
2. Heap space analysis
3. Permission analysis
4. Cost Analysis
5. Conclusions

Security Model

Starting assumption: at any point in time a set of permissions is available in the control flow.

Certain operations may:

- *grant* a permission; this increases the permission set
- *request* a permission; this
  - either checks presence — classical access control,
  - or checks presence and **consumes an instance** — billable events

A program is safe if for every execution trace the requested permissions are present (no checks fail).

Consumption of use-once resources is naturally modelled using *multisets* of permissions, so each permission has a *multiplicity*. Including multiplicity ∞ subsumes classical case.

Security Model

Starting assumption: at any point in time a set of permissions is available in the control flow.

Certain operations may:

- *grant* a permission; this increases the permission set
- *request* a permission; this
  - either checks presence — classical access control,
  - or checks presence and **consumes an instance** — billable events

A program is safe if for every execution trace the requested permissions are present (no checks fail).

Consumption of use-once resources is naturally modelled using *multisets* of permissions, so each permission has a *multiplicity*. Including multiplicity ∞ subsumes classical case.

Security Model

Starting assumption: at any point in time a set of permissions is available in the control flow.

Certain operations may:

- *grant* a permission; this increases the permission set
- *request* a permission; this
  - either checks presence — classical access control,
  - or checks presence and **consumes an instance** — billable events

A program is safe if for every execution trace the requested permissions are present (no checks fail).

Consumption of use-once resources is naturally modelled using *multisets* of permissions, so each permission has a *multiplicity*. Including multiplicity ∞ subsumes classical case.

Bulk Messaging in MIDP, currently

MIDlet code

```
msg.edit();
grp = addr_book.sel_grp();     // user selects recipients
for (addr in grp) {
  num = addr.get_mobile();
  msg.send(num);               // user re-confirms each recipient
}
```

- Hard to detect that re-confirmation is unnecessary:
  - Selection and sending may happen in different threads.
  - User should confirm number rather than address book entry.

API for Resource Managers

- Idea: reify permission sets into code for *explicit accounting*
- New API for **resource managers**, with three classes:
  - `Resource` — abstract class for resources; subclasses immutable
  - `Multiset` — for manipulating permission sets
  - `ResourceManager` — encapsulation of permission set
    Special grant/request methods instrument standard operations.
- API needs careful design: resource manager should be tamper-proof, new information flows limited.
- Compared with leaving permissions implicit:
  - Pros: program-level inspection, flexible static/dynamic policies.
  - Cons: new runtime checks, platform library extension

API for Resource Managers

- Idea: reify permission sets into code for *explicit accounting*
- New API for **resource managers**, with three classes:
  - `Resource` — abstract class for resources; subclasses immutable
  - `Multiset` — for manipulating permission sets
  - `ResourceManager` — encapsulation of permission set
    Special grant/request methods instrument standard operations.
- API needs careful design: resource manager should be tamper-proof, new information flows limited.
- Compared with leaving permissions implicit:
  - Pros: program-level inspection, flexible static/dynamic policies.
  - Cons: new runtime checks, platform library extension

API for Resource Managers

- Idea: reify permission sets into code for *explicit accounting*
- New API for **resource managers**, with three classes:
  - `Resource` — abstract class for resources; subclasses immutable
  - `Multiset` — for manipulating permission sets
  - `ResourceManager` — encapsulation of permission set
    Special grant/request methods instrument standard operations.
- API needs careful design: resource manager should be tamper-proof, new information flows limited.
- Compared with leaving permissions implicit:
  - Pros: program-level inspection, flexible static/dynamic policies.
  - Cons: new runtime checks, platform library extension

What we want to type

MIDlet with resource manager

```
msg.edit();                                 // user edits message
nums = addr_book.sel_nums();                // user selects recipients
Multiset r = new Multiset();
for (num in nums) {                         // gather resources
  r.add(Resource.from_string(num));
}
ResMgr m = new ResMgr();
r = m.enable(r);                            // user grants resources
if (r.is_empty()) {
  for (num in nums) {                       // send loop, pass mgr
    msg.send_rm(m, num);
  }
}
```

Simplified setting

- To study the type system, we use a small procedural language (close to Java bytecode), with special types and operations:
  - resource type `res` with constructors (`from_string`), equality
  - multiset type `mset` with multiset operations
  - abstract type `mgr` with no operations; copying prohibited.
- Statements in SSA form:

$$\begin{aligned} s ::= \;& \texttt{skip} \mid \texttt{assert}\ x_1\ \texttt{in}\ x_2 \mid y := e \mid s_1 ; s_2 \mid \\ & \texttt{if}\ x\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi} \mid p(x_1, \ldots, x_m | y_1, \ldots, y_n) \end{aligned}$$

- Procedure parameters partitioned by input/output mode.
- All `mgr`-variables will be linear (used exactly once).
- Assert checks $x_1 \subseteq x_2$, raises error $\lightning$ otherwise
- Also: built-in procedures for manipulating managers...

Simplified setting

- To study the type system, we use a small procedural language (close to Java bytecode), with special types and operations:
  - resource type `res` with constructors (`from_string`), equality
  - multiset type `mset` with multiset operations
  - abstract type `mgr` with no operations; copying prohibited.
- Statements in SSA form:

$$s ::= \texttt{skip} \mid \texttt{assert}\ x_1\ \texttt{in}\ x_2 \mid y := e \mid s_1; s_2 \mid \\ \texttt{if}\ x\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi} \mid p(x_1, \ldots, x_m | y_1, \ldots, y_n)$$

- Procedure parameters partitioned by input/output mode.
- All `mgr`-variables will be linear (used exactly once).
- Assert checks $x_1 \subseteq x_2$, raises error $\lightning$ otherwise
- Also: built-in procedures for manipulating managers. . .

Built-in procedures

- $\texttt{empty}(|m : \texttt{mgr})$
  - Set m to the empty multiset.
- $\texttt{enable}(r_1 : \texttt{mset}|m : \texttt{mgr}, r_2 : \texttt{mset})$
  - Ask user/policy to grant resources r_1.
  - Place granted resources in m, denied resources in r_2.
- $\texttt{split}(m : \texttt{mgr}, r : \texttt{mset}|m_1 : \texttt{mgr}, m_2 : \texttt{mgr})$
  - Split off from m all resources contained in r.
  - Place split off resources in m_2, remaining resources in m_1.
- $\texttt{join}(m_1 : \texttt{mgr}, m_2 : \texttt{mgr}|m : \texttt{mgr})$
  - Set m to the sum of resources in m_1 and m_2.
- $\texttt{consume}(m : \texttt{mgr}|)$
  - No-op (required due to strict linearity).

Example in simple language

Main procedure

```
bulk_send(msg:string, nums:string[] |) {
  r,r':mset;
  m,m':mgr;
  b:bool;

  res_from_nums(nums | r);
  enable(r | m, r');
  b := r' = ∅;
  if b
  then msg_send(msg, nums, m | m');
        assert m' in r';
        consume(m' |)
  else consume(m |)
  fi
}
```

Example: Bulk Messaging

Send loop — body

```
msg_send'(msg:string, nums:string[], m:mgr, i:int | m':mgr) {
  b:bool;
  num:string;
  i':int;
  m'':mgr;
  b := i < #nums;
  if b
  then num := nums[i];
       send_rm(msg, num, m | m'');
       i' := i+1;
       msg_send'(msg, nums, m'', i' | m')
  else move(m | m')
  fi
}
```

Example: Bulk Messaging

Instrumented send procedure

```
send_rm(msg:string, num:string, m:mgr | m':mgr) {
  r:mset;
  m'':mgr;
  r := {|from_string(num):1|};
  split(m, r | m', m'');
  assert r in m'';
  send(msg, num |);
  consume(m'' |)
}
```

Effect types

- Constraints Φ, Ψ are quantified FO formulae over expressions
- $\Phi \rightarrow \Psi$ is a pre-post constraint "effect" type:
  - for a statement, no defined variables allowed in input constraint Φ
  - for a procedure, no output variables allowed in input constraint Φ

The main judgement $\Gamma \vdash s : \Phi \rightarrow \Psi$

says for all executions of s starting in a state α with $\alpha \models \Phi$,

- the error $\lightning$ will not be raised, and
- if the execution terminates in a state β then $\beta \models \Phi \wedge \Psi$.

the judgement is relative to a program Prog and an effect environment Eff.

Effect types

- Constraints Φ, Ψ are quantified FO formulae over expressions
- $\Phi \longrightarrow \Psi$ is a pre-post constraint "effect" type:
  - for a statement, no defined variables allowed in input constraint Φ
  - for a procedure, no output variables allowed in input constraint Φ

The main judgement $\Gamma \vdash s : \Phi \longrightarrow \Psi$

says for all executions of s starting in a state α with $\alpha \models \Phi$,

- the error $\lightning$ will not be raised, and
- if the execution terminates in a state β then $\beta \models \Phi \wedge \Psi$.

the judgement is relative to a program Prog and an effect environment Eff.

Effect types for built-in procedures

$\texttt{empty}(|m : \texttt{mgr})$

- $\text{Eff}(\texttt{empty}) = \top \to m = \emptyset$

$\texttt{enable}(r_1 : \texttt{mset}|m : \texttt{mgr}, r_2 : \texttt{mset})$

- $\text{Eff}(\texttt{enable}) = \top \to r_1 = m \uplus r_2$

$\texttt{split}(m : \texttt{mgr}, r : \texttt{mset}|m_1 : \texttt{mgr}, m_2 : \texttt{mgr})$

- $\text{Eff}(\texttt{split}) = \top \to m_2 = m \cap r \wedge m = m_1 \uplus m_2$

$\texttt{join}(m_1 : \texttt{mgr}, m_2 : \texttt{mgr}|m : \texttt{mgr})$

- $\text{Eff}(\texttt{join}) = \top \to m = m_1 \uplus m_2$

$\texttt{consume}(m : \texttt{mgr}|)$

- $\text{Eff}(\texttt{consume}) = \top \to \top$

Typing Rules

skip

$$\overline{\Gamma \vdash \texttt{skip} : \top \to \top}$$

assertion

$$\overline{\Gamma \vdash \texttt{assert}\ x_1\ \texttt{in}\ x_2 : x_1 \subseteq x_2 \to \top}$$

assignment

$$\overline{\Gamma \vdash y := e : \top \to y = e}$$

procedure call

$$\frac{\mathsf{Prog}(p) = p(x_1 : \sigma_1, \ldots, x_m : \sigma_m | y_1 : \tau_1, \ldots, y_n : \tau_n)[\{\ldots\}] \quad \mathsf{Eff}(p) = \Phi \rightarrow \Psi}{\Gamma \vdash p(x'_1, \ldots, x'_m | y'_1, \ldots, y'_n) : \Phi' \rightarrow \Psi'}$$

sequential composition

$$\frac{\Gamma \vdash s_1 : \Phi \rightarrow \Psi_1 \qquad \Gamma \vdash s_2 : \Phi \wedge \Psi_1 \rightarrow \Psi_2}{\Gamma \vdash s_1; s_2 : \Phi \rightarrow \Psi_1 \wedge \Psi_2}$$

conditional branching

$$\frac{\Gamma \vdash s_1 : z \wedge \Phi \rightarrow \Psi \qquad \Gamma \vdash s_2 : \neg z \wedge \Phi \rightarrow \Psi}{\Gamma \vdash \mathtt{if}\ z\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2\ \mathtt{fi} : \Phi \rightarrow \Psi}$$

weakening

$$\frac{\Gamma \vdash s : \Phi \longrightarrow \Psi \qquad \begin{array}{c} \Phi' \models \Phi \\ (\Phi \Rightarrow \Psi) \models (\Phi' \Rightarrow \Psi') \end{array}}{\Gamma \vdash s : \Phi' \longrightarrow \Psi'}$$

procedure body

$$\frac{\begin{array}{c} \mathrm{Prog}(p) = p(x_1 : \sigma_1, \ldots, x_m : \sigma_m | y_1 : \tau_1, \ldots, y_n : \tau_n)\{z_1 : \rho_1, \ldots, z_l : \rho_l; s\} \\ \Gamma = x_1 : \sigma_1, \ldots, x_m : \sigma_m, y_1 : \tau_1, \ldots, y_n : \tau_n, z_1 : \rho_1, \ldots, z_l : \rho_l \\ \mathrm{Eff}(p) = \Phi \longrightarrow \Psi \\ \Gamma \vdash s : \Phi \longrightarrow \Psi \end{array}}{\mathrm{Eff} \vdash p}$$

Type soundness

Big-step semantics $\alpha \vdash s \rhd \beta$

- there is an execution of statement s starting in state α and terminating in state β.

Soundness Theorem

If

- $\mathrm{Eff} \vdash p$ for all procedures in the program
- $\Gamma \vdash s : \Phi \longrightarrow \Psi$, and
- $\alpha \vdash s \rhd \beta$

then $\alpha \neq \lightning \wedge \alpha \models \Phi$ implies $\beta \neq \lightning \wedge \beta \models \Phi \wedge \Psi$.

Type soundness

Big-step semantics $\alpha \vdash s \rhd \beta$

- there is an execution of statement s starting in state α and terminating in state β.

Soundness Theorem

If

- $\mathsf{Eff} \vdash p$ for all procedures in the program
- $\Gamma \vdash s : \Phi \longrightarrow \Psi$, and
- $\alpha \vdash s \rhd \beta$

then $\alpha \neq \lightning \wedge \alpha \models \Phi$ implies $\beta \neq \lightning \wedge \beta \models \Phi \wedge \Psi$.

Erasing resource managers from well-typed programs

- Erasure: $\texttt{mgr}^\circ = \texttt{unit}$, $(\texttt{assert}\ x_1\ \texttt{in}\ x_2)^\circ = \texttt{skip}$.
- After erasure, `enable` retains user/policy interaction:
  - $\text{Prog}^\circ(\texttt{enable}) = \texttt{enable}(r_1 : \texttt{mset} | m : \texttt{unit}, r_2 : \texttt{mset})$
  - $\text{Eff}^\circ(\texttt{enable}) = \top \rightarrow r_2 \subseteq r_1$.

Erasure Theorem

If

- $\text{Eff} \vdash p$ for all procedures in the program Prog,
- $\Gamma \vdash s : \Phi \rightarrow \Psi$, and
- $\alpha \neq ⚡ \wedge \alpha \models \Phi$

then for all states β, $\alpha \vdash s \triangleright \beta \Leftrightarrow \alpha^\circ \vdash s^\circ \triangleright \beta^\circ$.

Corollary

Resource managers cannot be used as covert channels (in the sense of information flow).

Erasing resource managers from well-typed programs

- Erasure: $\mathtt{mgr}^\circ = \mathtt{unit}$, $(\mathtt{assert}\ x_1\ \mathtt{in}\ x_2)^\circ = \mathtt{skip}$.
- After erasure, enable retains user/policy interaction:
  - $\mathrm{Prog}^\circ(\mathtt{enable}) = \mathtt{enable}(r_1 : \mathtt{mset} | m : \mathtt{unit}, r_2 : \mathtt{mset})$
  - $\mathrm{Eff}^\circ(\mathtt{enable}) = \top \rightarrow r_2 \subseteq r_1$.

Erasure Theorem

If

- $\mathrm{Eff} \vdash p$ for all procedures in the program Prog,
- $\Gamma \vdash s : \Phi \rightarrow \Psi$, and
- $\alpha \neq \lightning \wedge \alpha \models \Phi$

then for all states β, $\alpha \vdash s \rhd \beta \Leftrightarrow \alpha^\circ \vdash s^\circ \rhd \beta^\circ$.

Corollary

Resource managers cannot be used as covert channels (in the sense of information flow).

Erasing resource managers from well-typed programs

- Erasure: $\mathtt{mgr}^\circ = \mathtt{unit}$, $(\mathtt{assert}\ x_1\ \mathtt{in}\ x_2)^\circ = \mathtt{skip}$.
- After erasure, enable retains user/policy interaction:
  - $\mathrm{Prog}^\circ(\mathtt{enable}) = \mathtt{enable}(r_1 : \mathtt{mset} | m : \mathtt{unit}, r_2 : \mathtt{mset})$
  - $\mathrm{Eff}^\circ(\mathtt{enable}) = \top \to r_2 \subseteq r_1$.

Erasure Theorem

If

- $\mathrm{Eff} \vdash p$ for all procedures in the program Prog,
- $\Gamma \vdash s : \Phi \longrightarrow \Psi$, and
- $\alpha \neq \lightning \wedge \alpha \models \Phi$

then for all states β, $\alpha \vdash s \rhd \beta \Leftrightarrow \alpha^\circ \vdash s^\circ \rhd \beta^\circ$.

Corollary

Resource managers cannot be used as covert channels (in the sense of information flow).

Outline

1 Resource control overview

2 Heap space analysis

3 Permission analysis

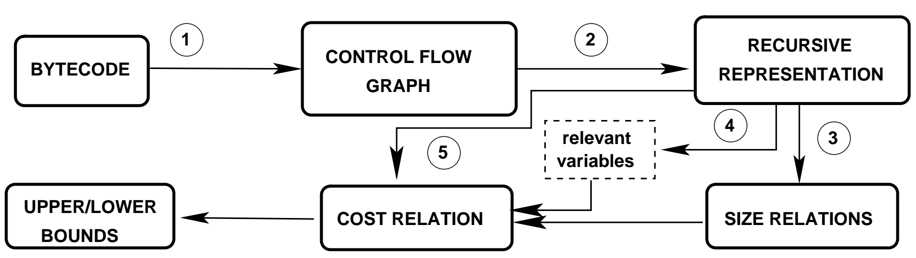
4 Cost Analysis

5 Conclusions

Bytecode cost analysis: motivations

- Cost analysis has been intensively studied for *declarative* and *high-level* imperative programming languages.
- Traditionally, cost analysis has been formulated at the source level. Mobius PCC framework requires analysis of compiled bytecode.
- Receiver wants cost information to reject code with too large cost requirements and accept code with established requirements
- Mobius technology: automatic approach to cost analysis of Java bytecode which statically generates **cost relations** as functions of input data size.

Method for cost analysis



1. Standard
2. Flatten stack; SSA
3. Global fixed point analysis
4. Slice away irrelevant variables
5. Costs in recursive form

A Matrix Multiplication Example

```
public int[][] mmult(int[][] A, int[][] B) {
  int c=A.length;  int r=A[0].length;  int[][] C=new int[r][c];
  for(int i=0; i<r; i++)
    for(int j=0; j<c; j++)
      for(int k=0; k<c; k++)
        C[i][j] = C[i][j] + (A[i][k] * B[k][j]);
  return C; }
```

The cost equations:

$$
\begin{aligned}
&\mathrm{mmult}(r,c) = 16 + D_1(r',c',i), \{\}, \{r'=r, c'=c, i=0\} \\
&D_1(r,c,i) = 3, \{i \geq r\}, \{\} \\
&D_1(r,c,i) = 7 + D_2(c',j) + D_1(r',c'',i'), \{i < r\}, \\
&\qquad\qquad\qquad\qquad \{c'=c, j=0, r'=r, c''=c, i'=i+1\} \\
&D_2(c,j) = 3, \{j \geq c\}, \{\} \\
&D_2(c,j) = 7 + D_3(c',k) + D_2(c'',j'), \{j < c\}, \{k=0, j'=j+1, c'=c, c''=b\} \\
&D_3(c,k) = 3, \{k \geq c\}, \{\} \\
&D_3(c,k) = 27 + D_3(c',k'), \{k < c\}, \{c'=c, k'=k+1\}
\end{aligned}
$$

The closed form: $19 + r(10 + c(10 + 27c)) = 27rc^2 + 10rc + 10r + 19.$

Summary of cost analysis

- A cost analysis framework for Java bytecode, including:
  - unstructured control flow
  - operand stack
  - exceptions, objects, and virtual method invokation
- It produces cost relations for methods in terms of the size of their input arguments:
  - integer values for numeric arguments
  - length for arrays
  - path-length for objects
- We have implemented the framework
  - still a prototype
  - but promising results
- It is parametric w.r.t. the cost model:
  - we have successfully applied to measure number of bytecode instructions executed.
  - currently, we are applying it to measure heap consumption.

Outline

1. Resource control overview
2. Heap space analysis
3. Permission analysis
4. Cost Analysis
5. Conclusions

Summary

- Several kinds of resource: machine intrinsic; explicit in code; external.
- Type-based analysis methods: automatic, efficient, certifying.
- Certification approaches: base logic proofs; admissible embedding of type systems; meta properties.
  - *erasure* meta-property for resource managers: certified code may skip checks
- Particular resources: heap space, permissions, execution time.
- Work continuing in Mobius: improving analyses and automation, handling rest of Java bytecode where not yet considered, implementing certification mechanisms; unifying approaches.

Credits and references

- E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini.
  Cost analysis of Java bytecode.
  In *ESOP*, pages 157–172, 2007.
- F. Besson, G. Dufay, and T. Jensen.
  A formal model of access control for mobile interactive devices.
  In *ESORICS*, Hamburg, September 2006. Springer Verlag.
- Lennart Beringer and Martin Hofmann.
  Type system for constant heap space.
  In *Mobius Deliverable D2.1*. 2006.
- Patrick Maier, D. Aspinall, and I. Stark.
  Explicit accounting of resources using resource managers.
  Technical Report EDI-INF-RR-0859, University of Edinburgh, 2006.