

A Systematic Approach to Static Access Control

François Pottier (INRIA),
Christian Skalka and Scott Smith (JHU)

Motivations

- 1 To develop types as tools for *statically* analyzing general purpose PL's with *security features*
 - Types provide declarative security mechanism
 - Types can improve efficiency, eagerly detect security errors
- 2 To present an instance of a general *transformational* approach to type system development
 - Approach allows re-use of established soundness, type inference results
 - Instance improves upon a previous system developed by Skalka and Smith

Part 1

- 1 Review Skalka and Smith's work on *Java Stack Inspection*
- 2 Review Pottier's *transformational* approach to developing type systems
- 3 Formally define a stack inspection language model λ_{sec}
- 4 Define a transformation of λ_{sec} into a target language λ_{set}

Types for PL Security

“Static Enforcement of Security with Types”, Skalka & Smith ICFP00

Java JDK1.2 provides a language-based mechanism for defining and enforcing *access control* policies:

- All programs are signed by the code *owner* (principal)
- Local policy maps owners to sets of authorized local privileges in ACL's
- Privileges may be explicitly enabled and checked dynamically by *Stack Inspection*

Stack Inspection

Each program call-stack frame is annotated with identity of frame owner and any privileges activated within frame:

- $\text{enablepriv}(r)$ places privilege r on current stack frame if stack frame owner is authorized for r
- $\text{checkpriv}(r)$ initiates stack inspection; stack frames are searched from most to least recent:
 - **fail** if a stack frame owned by a principal unauthorized for r is encountered
 - otherwise **succeed** if r is found

Stack inspection example

Assuming Joe is authorized for `print` privilege:

(* owned by System *)

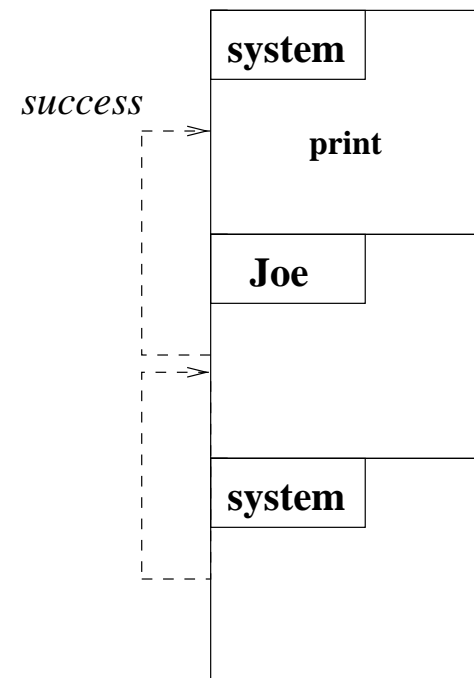
$enablePrint(g, x) = \{enablepriv(\mathbf{print}); g(x)\}$

$safePrint(f) = \{checkpriv(\mathbf{print}); print(f)\}$

(* owned by Joe *)

$joeProg(f) = \{safePrint(f)\}$

$enablePrint(joeProg, file)$



Stack inspection example

Assuming Joe is *not* authorized for print privilege:

(* owned by System *)

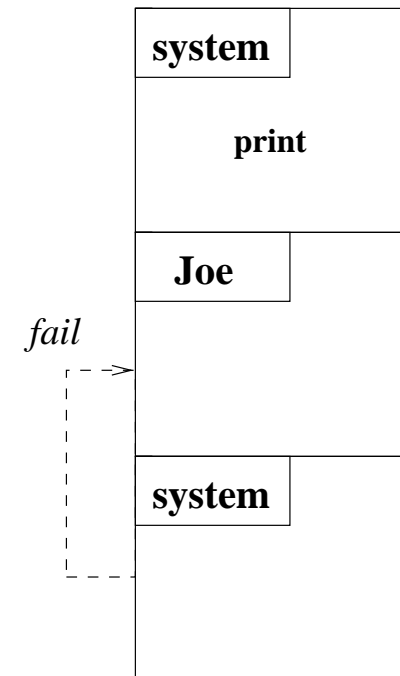
$enablePrint(g, x) = \{enablepriv(\mathbf{print}); g(x)\}$

$safePrint(f) = \{checkpriv(\mathbf{print}); print(f)\}$

(* owned by Joe *)

$joeProg(f) = \{safePrint(f)\}$

$enablePrint(joeProg, file)$



Static Stack Inspection

Our contribution: In a simplified language, types can be used to statically enforce the stack inspection model:

- Types provide declarative security mechanism
- Static approach improves runtime efficiency by eliminating need for dynamic checks

However: Language must be extended, monomorphic types, constraint-based types hard to read.

Transformational Approach

“Information Flow Inference for Free”, Pottier & Conchon ICFP00

Type system for expressions e in a novel *source language* is obtained by transformation $\langle e \rangle$:

- $\langle e \rangle$ is a term in a familiar *target language* with sound type system
- Transformation preserves meaning: if $e \rightarrow^* v$ then $\langle e \rangle \rightarrow^* \langle v \rangle$
- Types for expressions e obtained from target type system: if $\langle e \rangle : \tau$ then $e : \tau$

Transformational Approach

Correctness of term transformation $\langle e \rangle$ yields a source language type system “for free”:

- Soundness results *automatically* apply to types $e : \tau$ in source language
- If $\langle e \rangle : \tau$ can be inferred, compose transformation and type inference to infer $e : \tau$
- Method may yield insight into semantics and/or desired structure of direct types for source language

Transforming Stack Inspection

The transformational approach can be used to develop type systems for static stack inspection:

- Allows re-use of well-known system (polymorphic *row types*), along with soundness results
- Allows re-use of existing *unification* algorithm— types more readable
- Non-trivial language extensions with little theoretical overhead

Source language: λ_{sec}

The language is a lambda calculus extended with constructs for modeling stack inspection:

$p \in \mathcal{P}, P \subseteq \mathcal{P}$	<i>principals</i>
$r \in \mathcal{R}, R \subseteq \mathcal{R}$	<i>resources</i>
$\mathcal{A} \in \mathcal{P} \rightarrow 2^{\mathcal{R}}$	<i>access credentials</i>
$v ::= \lambda x.f$	<i>values</i>
$f ::= p.e$	<i>signed expressions</i>
$e ::= x \mid \lambda x.f \mid e e \mid \text{let } x = e \text{ in } e \mid$ $\text{letpriv } r \text{ in } e \mid \text{checkpriv } r \text{ for } e \mid$ $\text{testpriv } r \text{ then } e \text{ else } e \mid f$	<i>expressions</i>

The `testpriv` construct allows *testing* the presence or absence of r in the current environment.

λ_{sec} examples

$$\mathcal{A} = \{p_0 \mapsto \emptyset, p \mapsto \{r\}\}$$

$\text{test}_r = \text{testpriv } r \text{ then } v_1 \text{ else } v_2$ v_1, v_2 closed

$\text{check}_r = \lambda x.p.\text{checkpriv } r \text{ for } x$

$\text{id} = \lambda x.p.x$

$\text{test}_r \rightarrow^* v_2$

letpriv r in $\text{test}_r \rightarrow^* v_1$

$\text{check}_r \text{ id}$ *goes wrong!*

Stack inspection formalization

Stacks are defined in terms of *evaluation contexts*:

$$E ::= [] \mid E e \mid v E \mid \text{let } x = E \text{ in } e \mid \text{letpriv } r \text{ in } E \mid p.E$$

Instead of an explicit semantic construct, stacks are inferred from the evaluation context:

$$\begin{array}{ll} \text{stack}([]) = \varepsilon & \text{stack}(E e) = \text{stack}(E) \\ \text{stack}(v E) = \text{stack}(E) & \text{stack}(\text{let } x = E \text{ in } e) = \text{stack}(E) \\ \text{stack}(\text{letpriv } r \text{ in } E) = r.\text{stack}(E) & \text{stack}(p.E) = p.\text{stack}(E) \end{array}$$

Stacks are as they would exist for expressions in the hole.

Stack example

$$E = p'.\text{letpriv } r \text{ in } p.[] \quad \text{stack}(E) = p'.r.p$$

$$id = \lambda x.p.x$$

$$check_r = \lambda x.p.\text{checkpriv } r \text{ for } x$$

$$letp_r = \lambda f.p'.\text{letpriv } r \text{ in } f \text{ } id$$

$$letp_r \text{ } check_r \rightarrow p'.\text{letpriv } r \text{ in } check_r \text{ } id$$

$$\rightarrow p'.\text{letpriv } r \text{ in } p.\text{checkpriv } r \text{ for } id$$

$$p'.\text{letpriv } r \text{ in } p.\text{checkpriv } r \text{ for } id = E[\text{checkpriv } r \text{ for } id]$$

Stack inspection algorithm

Stack inspection is a decidable relation on stacks S and privileges r , written $S \vdash r$. Informally, it works much the same as in Java:

- Inspection begins on the right end of the stack
- $S \not\vdash r$ if an unauthorized principal is encountered
- $S \vdash r$ if r is found, and next principal to the left is authorized for r

For example, assuming $r \in \mathcal{A}(p) \wedge \mathcal{A}(p')$:

$$p'.r.p \vdash r$$

$$E = p'.\text{letpriv } r \text{ in } p.[] \quad \text{stack}(E) = p'.r.p$$

$$\text{stack}(E) \vdash r$$

$E[\text{checkpriv } r \text{ for } id]$ should be safe...

Operational semantics for λ_{sec}

Semantics combine term reduction and stack inspection:

$$\begin{array}{llll} E[(\lambda x.f) v] & \rightarrow & E[f[v/x]] & \\ E[\text{let } x = v \text{ in } e] & \rightarrow & E[e[v/x]] & \\ E[\text{checkpriv } r \text{ for } e] & \rightarrow & E[e] & \text{if } \text{stack}(E) \vdash r \\ E[\text{testpriv } r \text{ then } e_1 \text{ else } e_2] & \rightarrow & E[e_1] & \text{if } \text{stack}(E) \vdash r \\ E[\text{testpriv } r \text{ then } e_1 \text{ else } e_2] & \rightarrow & E[e_2] & \text{if } \text{stack}(E) \not\vdash r \\ E[\text{letpriv } r \text{ in } v] & \rightarrow & E[v] & \\ E[p.v] & \rightarrow & E[v] & \end{array}$$

What target language?

Target language depends on translation we choose.

We use Wallach's *security-passing-style (SPS)* conversion:

- Security information is *passed down the stack* as an explicit function parameter, rather than maintained in stack annotations
 - Security information represented as sets of activated privileges
 - enabling a privilege adds it to the activated privileges, checking a privilege checks its membership
 - $f(x)$ is transformed to $f(x, s)$, where s denotes currently activated privileges *intersected with* the privileges authorized to owner of f

Target language: λ_{set}

Our target language includes constructs for defining sets, with set membership test, union and intersection operations:

$$\begin{array}{ll} e ::= x \mid v \mid e e \mid \text{let } x = e \text{ in } e & \text{expressions} \\ v ::= \lambda x. e \mid R \mid .r \mid ?_r \mid \vee_R \mid \wedge_R & \text{values} \end{array}$$

$?_r$ allows for images of testpriv expressions in the transformation. The operational semantics is simple:

$$\begin{array}{lll} (\lambda x. e) v & \rightarrow & e[v/x] \\ \text{let } x = v \text{ in } e & \rightarrow & e[v/x] \\ R.r & \rightarrow & R \quad \text{if } r \in R \\ ?_r R & \rightarrow & \lambda f. \lambda g. (f R) \quad \text{if } r \in R \\ ?_r R & \rightarrow & \lambda f. \lambda g. (g R) \quad \text{if } r \notin R \\ R_1 \vee R_2 & \rightarrow & R_1 \cup R_2 \\ R_1 \wedge R_2 & \rightarrow & R_1 \cap R_2 \\ E[e] & \rightarrow & E[e'] \quad \text{if } e \rightarrow e' \end{array}$$

The λ_{sec} -to- λ_{set} transformation (highlights)

Our “security passing style” transformation is defined (in part) as follows:

$$\begin{aligned} \llbracket \lambda x. f \rrbracket_p &= \lambda x. \lambda s. \llbracket f \rrbracket \\ \llbracket e_1 e_2 \rrbracket_p &= \llbracket e_1 \rrbracket_p \llbracket e_2 \rrbracket_p s \\ \llbracket \text{letpriv } r \text{ in } e \rrbracket_p &= \text{let } s = s \vee (\{r\} \cap \mathcal{A}(p)) \text{ in } \llbracket e \rrbracket_p \\ \llbracket \text{checkpriv } r \text{ for } e \rrbracket_p &= \text{let } _ = s.r \text{ in } \llbracket e \rrbracket_p \\ \llbracket \text{testpriv } r \text{ then } e_1 \text{ else } e_2 \rrbracket_p &= ?_r s (\lambda s. \llbracket e_1 \rrbracket_p) (\lambda s. \llbracket e_2 \rrbracket_p) \\ \llbracket p.e \rrbracket &= \text{let } s = s \wedge \mathcal{A}(p) \text{ in } \llbracket e \rrbracket_p \end{aligned}$$

Free occurrences of s are assigned \emptyset by the top-level transformation function $\langle e \rangle$, where $\langle e \rangle = \llbracket e \rrbracket_{p_0}[\emptyset/s]$.

Transformation example

$$check_r = \lambda x.p.checkpriv\ r\ \text{for } x$$

$$id = \lambda x.p.x$$

$$\llbracket check_r \rrbracket_{p_0} = \lambda x.\lambda s.let\ s = s \wedge \{p\}\ \text{in } let_ = s.r\ \text{in } x$$

$$\llbracket id \rrbracket_{p_0} = \lambda x.\lambda s.let\ s = s \wedge \{p\}\ \text{in } x$$

$$(check_r\ id) = (\llbracket check_r \rrbracket_{p_0})(\llbracket id \rrbracket_{p_0})(\emptyset)$$

$$(check_r\ id) \rightarrow^* let\ s = \emptyset \wedge \{p\}\ \text{in } let_ = s.r\ \text{in } \llbracket id \rrbracket_{p_0}$$

Correctness of transformation

We establish the appropriate formal properties of our transformation:

Theorem: If $e \rightarrow^* v$, then $\langle e \rangle \rightarrow^* \langle v \rangle$. If e goes wrong, then $\langle e \rangle$ goes wrong. If e diverges, then $\langle e \rangle$ diverges.

This straightforward *syntactic* result allows us to apply type safety and inference in λ_{set} to λ_{sec} .

Part 2

- 1 Typing the target language λ_{set} :
 - (a) Review the $\text{HM}(X)$ system
 - (b) Define SETS, set types and constraints

- 2 Using $\text{HM}(\text{SETS})$ as a foundation for typing λ_{sec} :
 - (a) “Indirect” type safety *for free*
 - (b) Derive a “direct”, sound type system for λ_{sec}
 - (c) Type inference for λ_{sec}

Types for λ_{set}

We want a sound type system that will ultimately be applied to λ_{sec} :

- Types should track contents of sets
- $?_r$ expressions should be typed *accurately*
- Types should be readable; unification-based inference

A problem: no existing satisfactory type system! But we don't have to start from scratch...

The HM(X) system

“Type Inference with Constrained Types”. Odersky, Sulzmann and Wehr 1999

HM(X) is a generalized *type framework*:

- Let-polymorphism for core functional calculus is provided
- System is specialized by instantiating X with a *constraint system*:
 - Type grammars may be extended
 - Initial typing environment defined for new constants
 - Interpretation of a \leq rule defined
- Type inference provided, modulo constraint solution algorithm

The HM(X) system

The framework contains expressions, functions types, type schemes and basic constraints:

$e ::= x \mid \lambda x.e \mid ee \mid \text{let } x = e \text{ in } e$	<i>expressions</i>
$\tau ::= \alpha, \beta, \dots \mid \tau \rightarrow \tau$	<i>types</i>
$\sigma ::= \forall \bar{\alpha}[C].\tau$	<i>type schemes</i>
$C ::= \mathbf{true} \mid C \wedge C \mid \tau = \tau \mid \tau \leq \tau$	<i>constraints</i>

Type judgements are of the form $C, \Gamma \vdash e : \sigma$.

Constraint satisfiability:

- An *assignment* ρ is a mapping from types τ to their interpretation in a model
- C is *satisfiable* iff $\exists \rho$ such that C holds in the model under its interpretation ($\rho \vdash C$)
- Constraints C *entail* constraints D ($C \Vdash D$) iff $\rho \vdash C$ implies $\rho \vdash D$.

HM(X) type judgment rules (highlights)

$$\frac{\text{LET} \quad C, \Gamma \vdash e_1 : \sigma \quad C, (\Gamma; x : \sigma) \vdash e_2 : \tau}{C, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}$$

$$\frac{\text{APP} \quad C, \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad C, \Gamma \vdash e_2 : \tau_2}{C, \Gamma \vdash e_1 e_2 : \tau}$$

$$\frac{\text{SUB} \quad C, \Gamma \vdash e : \tau \quad C \Vdash \tau \leq \tau'}{C, \Gamma \vdash e : \tau'}$$

$$\frac{\forall \text{ELIM} \quad C, \Gamma \vdash e : \forall \bar{\alpha}[D]. \tau' \quad C \Vdash [\bar{\tau}/\bar{\alpha}]D}{C, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau'}$$

HM(X) example

Let W be the Herbrand constraint system over algebra of types τ :

- Interpret \leq as equality
- Subsumption becomes trivial
- Constraint solving is *unification*

Equating $\forall\alpha[\mathbf{true}].\tau$ with $\forall\alpha.\tau$ and the judgment $\mathbf{true}, \Gamma \vdash e : \tau$ with $\Gamma \vdash e : \tau$, the system $\text{HM}(W)$ is the Hindley-Milner system.

The constraint system SETS

We extend the basic HM(X) system with set types (specialized row types):

$$\begin{array}{ll}
 \rho & ::= \alpha, \beta, \dots \mid r : c ; \rho \mid \partial c & \text{rows} \\
 c & ::= \mathbf{Pre} \mid \mathbf{Abs} \mid \mathbf{Either} & \text{capabilities} \\
 \tau & ::= \alpha, \beta, \dots \mid \tau \rightarrow \tau \mid \{\rho\} \mid \rho & \text{types} \\
 C & ::= C \wedge C \mid \tau = \tau \mid \tau \leq \tau \mid \text{if } c \leq \tau \text{ then } \tau \leq \tau & \text{constraints}
 \end{array}$$

Set types describe sets by asserting which elements are present (**Pre**) and which are absent (**Abs**); letting $R = \{r_1, \dots, r_n\}$:

$$\begin{array}{l}
 R : \{r_1 : \mathbf{Pre}; \dots ; r_n : \mathbf{Pre}; \partial \mathbf{Abs}\} \\
 R : \{r_1 : \mathbf{Pre}; \dots ; r_n : \mathbf{Pre}; r_{n+1} : \mathbf{Abs}; \partial \mathbf{Abs}\}
 \end{array}$$

Initial bindings of λ_{set} primitives

For $R = \{r_1, \dots, r_n\}$ the notation $R : c$ denotes $r_1 : c; \dots r_n : c$ and $R : \bar{\gamma}$ denotes $r_1 : \gamma_1; \dots r_n : \gamma_n$.

$$\begin{aligned} .r & : \forall \beta. \{r : \mathbf{Pre}; \beta\} \rightarrow \{r : \mathbf{Pre}; \beta\} \\ \forall_R & : \forall \beta \bar{\gamma}. \{R : \bar{\gamma}; \beta\} \rightarrow \{R : \mathbf{Pre}; \beta\} \\ \wedge_R & : \forall \beta \bar{\gamma}. \{R : \bar{\gamma}; \beta\} \rightarrow \{R : \bar{\gamma}; \partial \mathbf{Abs}\} \\ ?_r & : \forall \alpha \beta \gamma. \{r : \gamma; \beta\} \rightarrow \\ & \quad (\{r : \mathbf{Pre}; \beta\} \rightarrow \alpha) \rightarrow (\{r : \mathbf{Abs}; \beta\} \rightarrow \alpha) \rightarrow \alpha \end{aligned}$$

For any $?_r e_1 e_2$, this binding requires that the security annotations and result type of its e_1 and e_2 are identical.

Typing $?_r$ more precisely

We may wish to provide more accurate typings of $?_r$ and ultimately testpriv expressions:

$$?_r : \forall \bar{\alpha} \bar{\beta} \gamma [C]. \{r : \gamma; \beta\} \rightarrow (\{r : \mathbf{Pre}; \beta_1\} \rightarrow \alpha_1) \rightarrow (\{r : \mathbf{Abs}; \beta_2\} \rightarrow \alpha_2) \rightarrow \alpha$$

$$\text{where } C = \text{if } \mathbf{Pre} \leq \gamma \text{ then } \beta \leq \beta_1 \wedge \text{if } \mathbf{Abs} \leq \gamma \text{ then } \beta \leq \beta_2 \\ \wedge \text{if } \mathbf{Pre} \leq \gamma \text{ then } \alpha_1 \leq \alpha \wedge \text{if } \mathbf{Abs} \leq \gamma \text{ then } \alpha_2 \leq \alpha$$

The systems \mathcal{S}_i^{rel}

We may also interpret \leq in different ways:

- As equality =
- As subtyping $<$:
 - Basic coercions **Pre** $<$: **Either**, **Abs** $<$: **Either**
 - *record*, \rightarrow , *trans* rules defined as usual

We can now define a *family of four systems*:

- Let $i \in \{1, 2\}$ denote our choice of bindings for $?_r$
- Let $rel \in \{=, <:\}$ denote our interpretation of \leq

\mathcal{S}_i^{rel} stands for a particular variation of HM(SETS)

λ_{set} type safety

λ_{set} enjoys type safety in any of $\mathcal{S}_i^{\text{rel}}$:

Theorem: If e is a λ_{set} expression and $C, \Gamma \vdash e : \sigma$ is a valid judgment in any $\mathcal{S}_i^{\text{rel}}$, then e does not go wrong.

The proof is straightforward:

- Prove subject reduction cases for new language constants (δ -typability)
- HM(X) provides the rest!

“Indirect” type safety for λ_{sec}

Type safety for λ_{sec} may be obtained as corollary of preceding:

Definition: Let e be in λ_{sec} ; then $C, \Gamma \vdash e : \sigma$ iff $C, \Gamma \vdash \langle e \rangle : \sigma$.

Theorem: If e is a λ_{sec} expression and $C, \Gamma \vdash e : \sigma$ is a valid judgment in any $\mathcal{S}_i^{\text{rel}}$, then e does not go wrong.

Proof is by type safety of $\mathcal{S}_i^{\text{rel}}$'s and correctness of the transformation $\langle e \rangle$.

- It's shorter, easier than a direct type safety proof for λ_{sec}

$\mathcal{S}_1^{\overline{=}}$ -derived types for λ_{sec}

Rather than typing λ_{sec} via transformation, we may wish to do so directly:

- Represent set type ζ of security environment (stack) directly in expression type
- Represent owner of expression in type

Derived type judgments are of the form $p, \zeta, \Gamma \vdash e : \sigma$, with constraint-free type schemes and:

Lemma: $p, \zeta, \Gamma \vdash e : \sigma$ holds iff **true**, $(\Gamma_1; \Gamma; s : \zeta) \vdash \llbracket e \rrbracket_p : \sigma$ holds.

\mathcal{S}_1^- -derived judgements (highlights)

$$\frac{\text{LETPRIV}^+ \quad p, \{r : \mathbf{Pre}; \rho\}, \Gamma \vdash e : \tau \quad r \in \mathcal{A}(p)}{p, \{r : \varphi; \rho\}, \Gamma \vdash \text{letpriv } r \text{ in } e : \tau}$$

$$\frac{\text{CHECKPRIV} \quad p, \{r : \mathbf{Pre}; \rho\}, \Gamma \vdash e : \tau}{p, \{r : \mathbf{Pre}; \rho\}, \Gamma \vdash \text{checkpriv } r \text{ for } e : \tau}$$

$$\frac{\text{TESTPRIV} \quad p, \{r : \mathbf{Pre}; \rho\}, \Gamma \vdash e_1 : \tau \quad p, \{r : \mathbf{Abs}; \rho\}, \Gamma \vdash e_2 : \tau}{p, \{r : \varphi; \rho\}, \Gamma \vdash \text{testpriv } r \text{ then } e_1 \text{ else } e_2 : \tau}$$

\mathcal{S}_1^- -derived function types

By definition of the transformation, every λ_{set} expression $\llbracket e \rrbracket_p$ of function type has a type of the form $\tau \rightarrow \zeta \rightarrow \tau'$, which we abbreviate $\tau \xrightarrow{\zeta} \tau'$:

ABS

$$\frac{\star, \zeta_2, (\Gamma; x : \tau_1) \vdash f : \tau_2}{p, \zeta_1, \Gamma \vdash \lambda x. f : \tau_1 \xrightarrow{\zeta_2} \tau_2}$$

APP

$$\frac{p, \zeta, \Gamma \vdash e_1 : \tau_2 \xrightarrow{\zeta} \tau \quad p, \zeta, \Gamma \vdash e_2 : \tau_2}{p, \zeta, \Gamma \vdash e_1 e_2 : \tau}$$

λ_{sec} type examples

$$id = \lambda x.p.x$$

$$check_r = \lambda x.p.\text{checkpriv } r \text{ for } x$$

$$enable_r = \lambda f.p.\lambda x.p.\text{letpriv } r \text{ in } f x$$

$$id : \forall \alpha \beta \gamma. \alpha \xrightarrow{\{r:\gamma; \beta\}} \alpha$$

$$check_r : \forall \alpha \beta. \alpha \xrightarrow{\{r:\mathbf{Pre}; \beta\}} \alpha$$

$$enable_r : \forall \dots . \left(\alpha_1 \xrightarrow{\{r:\mathbf{Pre}; s:\gamma_1; \partial\mathbf{Abs}\}} \alpha_2 \right) \xrightarrow{\{\beta_1\}} \left(\alpha_1 \xrightarrow{\{r:\gamma_2; s:\gamma_1; \beta_2\}} \alpha_2 \right)$$

“Direct” type safety for λ_{sec}

To establish λ_{sec} type safety in the derived system, we specialize a previously mentioned lemma:

Lemma: $p_0, \{\partial\mathbf{Abs}\}, \Gamma \vdash e : \sigma$ holds iff **true**, $(\Gamma_1; \Gamma) \vdash \langle e \rangle : \sigma$ holds.

The proof is easy, since the systems are closely related. λ_{sec} type safety is a corollary of this lemma and λ_{set} type safety:

Theorem: If $p_0, \{\partial\mathbf{Abs}\}, \Gamma \vdash e : \sigma$ holds then e does not go wrong.

- A *readable, concise* type system for static enforcement of the Stack Inspection model.

λ_{sec} type inference

The same approach may be used to provide *unification-based type inference* for λ_{set} and λ_{sec} types:

- General row unification algorithm by Didier Rémy *
- HM(X) provides type inference framework (constraint inference with constraint solver unspecified)
- To infer type of e :
 - compose transformation $(\lfloor e \rfloor)$ with HM(X) inference instantiated with row type unification

Direct type inference for λ_{sec} can easily be derived from this algorithm

* *in Theoretical Aspects Of Object-Oriented Programming. MIT Press, 1993*

Future work: the OWN rule

The OWN rule can lead to verbose types:

$$\frac{\text{OWN} \quad p, \{r_1 : \varphi_1; \dots; r_n : \varphi_n; \partial \mathbf{Abs}\}, \Gamma \vdash e : \tau \quad \mathcal{A}(p) = \{r_1, \dots, r_n\}}{\star, \{r_1 : \varphi_1; \dots; r_n : \varphi_n; \rho\}, \Gamma \vdash p.e : \tau}$$

If $\mathcal{A}(p)$ is large, types may contain lots of superfluous information.

- We have begun investigating *row-abbreviation* mechanisms to address this problem

Future work

- Dealing with testpriv:
 - Dynamic checks are still required (can't always predict which branches will be taken)
 - Soft typing
- Language extensions:
 - Parameterized privileges, e.g. `fileRead(filename)`
 - Exceptions, state and modules

Conclusion

- Types can be used for PL-based security:
 - Efficient, static stack inspection for access control
 - Declarative mechanism for expressing policies
- Transformational approach can be used for systematic development of our security type system:
 - Novel λ_{sec} system transformed into familiar λ_{set}
 - Transformational correctness and δ -typability of λ_{set} buys us λ_{sec} type safety
 - Row type unification yields inference of readable, expressive types

For more information: <http://www.cs.jhu.edu/~ces/work.html>