Constraint-based secure information flow inference for a Java-like language

Qi Sun
Stevens Institute of Technology
sunq@cs.stevens-tech.edu

David A. Naumann
Stevens Institute of Technology
naumann@cs.stevens-tech.edu

Anindya Banerjee
Kansas State University
ab@cis.ksu.edu

Abstract

We address the problem of checking programs written in a Java-like language, to ensure that they satisfy information flow policies like confidentiality and integrity in the presence of dynamic access control mechanisms like stack inspection. Security policy is specified using permission dependent security types. We present an inference algorithm that infers such security types in a modular manner. For an object-oriented language, this involves inference in the presence of libraries, i.e., classes parameterized by security levels. Moreover we show how modular inference is preserved in the presence of method inheritance and override. We discuss a prototype implementation and prove soundness of the algorithm.

1 Introduction

This paper addresses the problem of checking programs to ensure that they satisfy information flow policies like confidentiality and integrity *in the presence of dynamic access control mechanisms*. Confidentiality, for example, is an important requirement in several security applications – by itself, or as a component of other security policies (e.g., authentication). Even in security protocol design, e.g., web service protocols, confidentiality is considered a desirable property to enforce [Aba99]. Although impressive advances have been made in specifying static analyses for confidentiality, these have not seen much use. The correctness property of the analyses, called noninterference, is difficult to enforce and, moreover, the property fails in the presence of declassification.

In current practice, access control is widely used, not only at the level of operating systems, but also in *code components*. For example, the Java virtual machine and the .NET runtime system provide a dynamic access control mechanism – stack inspection – in which permissions are granted to components based on their origin, and a runtime mechanism checks permissions of code in the calling chain. In this setting, the need for access control arises because different components have different levels of trust and computation proceeds with trusted and less trusted components calling each other's methods. A typical design pattern is the following: untrusted code calls trusted code by enabling some of its permissions; trusted code checks that the permissions are indeed authorized for the untrusted caller before executing the code.

A drawback of access control is that it controls the release of information, but not the flow of information. Hence the question arises as to what is the connection between authorization of data access and the subsequent propagation of the data? Clearly, the intent is that authorization of access should not lead to violation of information flow policies like confidentiality: confidential data should not flow to public channels.

The primary contribution of previous work by Banerjee and Naumann [BN02, BN03b, BN03a] is the formalization of a type-based static analysis for information flow for a rich subset of a Java-like class-based object-oriented language, that takes dynamic access control into account. Following the pioneering work of Smith and Volpano, information flow policy is expressed by labelling of input and output channels with levels, e.g., low (L) and high (H)in a security lattice ($L \leq H$). The static analysis is given in the style of a security type system, where the chief novelty is that security policy is specified using permission-dependent security types. Consequently, the same method may have different security types depending on what permissions may be excluded in its calling context. The type checking rules account for calls into the trusted computing base that may return high security information, but which use access checks to ensure that only low security information is returned unless the caller has been given access.

The primary goal of this paper is the *inference* of security types for a Java-like language with primitives that permit dynamic access control via stack inspection. This leads to the following challenges: first, we demand inference of some, possibly all, security levels of fields in a class. This means that security types of fields will involve *level variables*. Because fields may be accessed and updated in method bodies (which are commands), the inference for

commands will involve level variables. Moreover, for each excluded permission set, we would like to infer the security levels for the arguments and result of a method. Hence the security type of a method may involve level variables.

Second, for practical reasons, we insist that our inference algorithm be *modular*. This means we have to carefully split the code into library classes (for which inference has already been performed) and the current analysis unit (for which inference is currently taking place). The current analysis unit may have mutually recursive class and method declarations and may contain multiple calls to library methods. A modular analysis must ensure that no library class is *re*-analyzed. This means that each library class must be *parameterized*, written $C{<}\alpha_1, \ldots, \alpha_n{>}$, over the level variables $\alpha_1, \ldots, \alpha_n$ mentioned in its field declarations. A method m declared in a parameterized library class can now be dispatched to in multiple ways depending on the ground instantiations of the parameters of the library class. There is yet another important aspect of a modular analysis. Because we are analyzing an object-oriented language, we need to consider how a modular analysis interacts with method inheritance and override. The current analysis unit can contain subclasses of a library class with some library methods overridden or inherited. Again, the analysis of the current unit should not entail analysis of the library class: we must ensure that the method signature of a library method is invariant with respect to subclassing.

Third, parameterized classes call for a language extension. The extended language is not Generic Java [BOSW98, IPW01]: we are not parameterizing over types but over security levels. Because method bodies are commands, assignment and field updates enforce *constraints* on the level variables in the method signature. For the inference algorithm to be tractable, we avoid polymorphic recursion [Myc84, Hen93] in method bodies.

Fourth, we need a proof of soundness of the inference algorithm with respect to the security type system. But the security type system only types a complete class table, i.e., a closed collection of class declarations, Our inference algorithm, on the other hand, constructs a library where each class is parameterized by the levels in its fields. For the inference algorithm to be sound, we consider any ground instantiation of the level variables occurring in the fields of each class. Then for any ground instantiation of the method signatures of each method in each class, if the constraints in the method signatures are satisfied, the resulting complete class table – bereft of all level variables – is well-formed in the security type system.

Overview. The contribution of this paper is to address all of the four points above. After discussing two simple examples in Section 2, we describe the language extended with parameterized classes in Section 3, explain the inference algorithm in Section 4 and give a proof of soundness of the algorithm in Section 5. Related work, including a preliminary implementation of the algorithm, is discussed in Section 6. Section 7 concludes.

2 Examples

Consider a class, IRS, as follows.

```
class IRS extends Object { // permissions = ∅
  int income;
  int tax(int salary){
    self.income := salary;
    result := self.income * 0.20; }}
```

The type of method *tax* in class *IRS*, written *mtype* (*tax*, *IRS*) is $int \rightarrow int$. Assuming that *income* has security level H, a possible *security type* for method *tax* in a context where no permissions are excluded, is $L, H{-}\langle H\rangle{\rightarrow}H$. This means if *tax* is called in a context where the level of the current object (i.e., the level of variable *self*) is L and the level of *salary* is at most H then the result level is at most H and (the H in the middle) only H fields may be updated during method execution. This security type can be verified using security type checking rules for method declaration and commands.

In security type inference, we wish to *infer* security types for the level of the field *income* and for the *tax* method. Let us fix the set of permissions that must be excluded in the caller's context to be $\varnothing$. We will assume that *IRS* is a well-formed class declaration. The level of *income* may be constant (e.g., L or H) or could be a variable that is a placeholder for the actual level of the field. In the sequel, we will let letters from the beginning of the Greek alphabet range over level variables.

Assume that all levels are unknown: that the level of *income* is α_1, the level of the *IRS* object is α_2, the level of *salary* is α_5 and the level of the return result is α_4. Finally, suppose that field of level at least α_3 can be assigned to during execution of *tax*. Informally, the intention is that for any ground instantiation of the field level α_1, and for any ground instantiations of the levels for the method, the method body should be typable in the security type system in Table 4. However, note that not all ground instantiations will lead to well-typed method declarations. For example, if salary has level H, then for the update *self.income* := *salary* to be well-typed, *income* cannot have level L. This means that constraints must be imposed between the α's in the inferred type for *tax*. And, to check typability of the method body, only those ground instantiations of the α's need be considered that satisfy the constraints.

For callers with the empty set of excluded permissions, our algorithm computes the type

$$(\alpha_2, \alpha_5 \text{–}\langle\alpha_3\rangle\!\rightarrow \alpha_4) \mid K$$

where K is a set of constraints; for example, $\alpha_3 \leq \alpha_1$ is a constraint appearing in K. Such a constraint makes sense because we have assumed (i) that the level of field *income* is α_1 and (ii) that the minimum level of the field must be α_3 for an assignment to occur.

The class *IRS* can now be converted into a library class, parameterized over α_1 and given a polymorphic, constrained method signature. This library class can be used multiple times in the analysis of another class, for instance, by instantiating α_1 with a ground level, say H. Consequently, all occurrences of α_1 appearing in the method signature are grounded to H. Note that in a separate instantiation of *IRS*$<\alpha_1>$, where α_1 is L, fresh copies of all other level variables appearing in the method signature must be generated.

```
class IRS<α1>extends Object { // permissions= ∅
  (int, α1) income;
  int tax(int salary){
    self.income := salary;
    result := self.income * 0.20; }}
```

Class *Inquiry* uses *IRS*$<\beta_1>$ as the type of field *emp*.

```
class Inquiry extends Object { // permissions = {audit}
  (IRS<β1>, β2) emp;
  (int, L) est; //actual tax
  bool overpay(){
    test audit then
    int tmp := emp.tax(1000);
    result := (tmp ≥ self.est)
    else result := true; }}
```

For inference, the level of *emp* is assumed to be β_2. We could have chosen the level of *emp* to be a constant, e.g., H. But suppose *IRS* contained a public *name* field with level L. Then it should be possible to access *emp*'s name publicly: this is facilitated by β_2. Note that the level of *est* is completely specified. Method *overpay* will have two interesting types, one for the case where no permissions are excluded and the other for the case where *audit* is excluded in the caller's context. They can be computed in a manner similar to that for method *tax*. In the former case, the type is $\alpha_1, () \text{–}\langle\alpha_2\rangle\!\rightarrow \alpha_3$ with some constraints K and in the latter case the type is $\alpha_4, () \text{–}\langle\alpha_5\rangle\!\rightarrow \alpha_6$ with the empty set of constraints. Some of the constraints in K generated by our inference algorithm are $\alpha_1 \leq \alpha_3$, $\beta_1 \leq \alpha_3$ and $\beta_2 \leq \alpha_3$. If the inquiry itself is secret, then the call to *overpay* should provide a secret answer. That this is the case can be seen from the constraint $\alpha_1 \leq \alpha_3$ with α_1, the level of the *Inquiry* object instantiated to H. Similarly, if the income is secret, then the method call should provide a secret answer. This is the purpose of constraint $\beta_1 \leq \alpha_3$ with β_1 instantiated to H. Note that this leads to instantiation of the *IRS* object as *IRS*$<H>$, which means the level of the *income* field is H. Finally, if *emp* itself is H, then the constraint $\beta_2 \leq \alpha_3$ forces the level of the return result to be H as expected.

3 Language

This section defines a sequential class-based language similar to the one used in our previous work [BN03a, BN03b], but with some improvements. The main improvement is that classes and methods may be polymorphic in levels. In this way, a library class can be used in more than one way. In this paper we make an explicit separation between a library and a collection of additional classes that are based on the library.

First, some terms: a *unit* is a collection of class declarations. A *closed unit* is a collection of class declarations that is well formed as a complete program, that is, it is a class table. The library is a closed unit from which what we need is its parameterized type signature, encoded in some auxiliary functions defined later. A program based on a library can consist of several classes which extend and use library classes and which may be mutually recursive. We use the term *analysis unit* for the classes to which the inference algorithm is applied. Due to mutual recursion, several classes may have to be considered together. An analysis unit must be well formed in the sense that the union of it with the library should form a closed unit.

Language. In this section we define the syntax for library units and also adapt the security typing rules from our previous work to the present language. Essentially, a polymorphic library is typable if all of its ground instances are.

A small difference between the security rules and those of our previous work is that the rules for `test` , which are the key to permission-dependent typing, have been made more precise. Another difference is that we had allowed each method m, C to be given a set $smtypes(m, C)$ of types. But as discussed in [BN03a], subsumption can be used to minimize a set. In particular, two types for a given permission set P can be combined, without loss of information, into a single type for P. Thus we consider method types to be indexed on P.

The grammar is in table 1. Although identifiers with overlines indicate lists, some of the formal definitions assume singletons to avoid unilluminating complication.

Since the problem we want to address is secure information flow, all the programs are assumed to be well formed as ordinary code; for this purpose, all levels are removed, including class parameters $<\overline{\lambda}>$. Typing rules for

Table 1. Language grammar

T	::=	$\texttt{bool} \mid C$ (where C ranges over $ClassName$)
κ	::=	$\texttt{H} \mid \texttt{L}$ (level constants)
λ	::=	$\alpha \mid \kappa$ (level variable, constant))
U	::=	$T\texttt{<}\overline{\lambda}\texttt{>}$
CL	::=	$\texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{extends}\ U\{(\overline{U},\overline{\lambda})\ \overline{f};\ \overline{M}\}$
M	::=	$U\ m\ (\overline{U}\ \overline{x})\{S\}$
S	::=	$x := e \mid e.f := e \mid x := \texttt{new}\ U()$
		$\mid x := e.m(\overline{e}) \mid S; S$
		$\mid U\ x := e\ \texttt{in}\ S \mid \texttt{if}\ e\ \texttt{then}\ S\ \texttt{else}\ S$
		$\mid \texttt{enable}\ P\ \texttt{in}\ S$ (enable permission set P)
		$\mid \texttt{test}\ P\ \texttt{then}\ S\ \texttt{else}\ S$ (test permissions P)
e	::=	$x \mid \texttt{null} \mid \texttt{true} \mid e.f \mid e == e \mid e\ \texttt{is}\ U \mid (U)e$

our Java-like language can be found in our previous papers [BN03a, BN03b]. It suffices to recall that a collection of class declarations, called a *class table*, is treated as a function CT so that $CT(C)$ is the code for class C. Moreover, $fields\ C$ is the field declarations for C and $mtype(m, C)$ gives the parameter and return types for m declared or inherited in C. Subtyping is invariant: $D \leq C$ implies $mtype(m, C) = mtype(m, D)$.

We assume given a finite set of permissions and an access policy, $Auth$, so that $Auth(C)$ is the static set of permissions that code in class C may enable.

Classes are parameterized with security variables. We use T to represent an ordinary data type, while U, W, R range over parameterized class types, which take the form $T\texttt{<}\overline{\lambda}\texttt{>}$. Declaration of a parameterized class binds some variables $\overline{\lambda}$ in the types of the superclass and fields:

$$\texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{extends}\ D\texttt{<}\overline{\lambda}\texttt{>}\{(\overline{U},\overline{\lambda'})\ \overline{f};\overline{M}\}$$

Thus it must satisfy a *well formedness* condition: $\overline{\alpha} \supseteq var(\overline{\lambda}) \cup var(\overline{U}) \cup var(\overline{\lambda'})$.

Field types, including security label, can be retrieved by a given function $lsfield(f, U)$ which returns the appropriate type of field f from an instantiated class U. Thus for the declaration displayed above,

$$lsfield(f, C\texttt{<}\overline{\alpha}\texttt{>}) = (\overline{U}, \overline{\lambda'})$$

and for any $\overline{\lambda''}$ of the right length,

$$lsfield(f, C\texttt{<}\overline{\lambda''}\texttt{>}) = (\overline{U}, \overline{\lambda'})[\overline{\alpha} \leftarrow \overline{\lambda''}]$$

We require that $lsfield(f, U)$ is defined iff $field(f, T)$ is defined, and moreover if $U \preceq U'$ and $lsfield(f, U')$ is defined, then $lsfield(f, U) = lsfield(f, U')$.

Method types, which depend on permissions and are polymorphic, need more delicate treatment. We assume a given function $lsmtype$ so that $lsmtype(m, U, P)$ returns the method m's signature and a set K of constraints in the form of inequalities between constants and variables. We require that $lsmtype(m, T\texttt{<}\overline{\kappa}\texttt{>}, P)$ is defined iff $mtype(m, T)$ is, and in that case the result takes the following form:

$$\lambda_0, (\overline{U}, \overline{\lambda}) \text{–}\langle\lambda_1\rangle\!\rightarrow (R, \lambda_2) | K$$

This judgement expresses the following policy:

- if the information in "self" is at most λ_0
- and the information in parameters is at most $\overline{\lambda}$ with type $\overline{U}$
- then any fields written are at level λ_1 or higher
- and the result level is at most λ_2, with type R
- provided that the constraints K are satisfied.

We also require *invariance under subclassing*: if class U extends R and m is declared in R, then $lsmtype(m, U, P) = lsmtype(m, R, P)$ (regardless of whether m is inherited or overridden in U.) Because $lsmtype$ is invariant under subclassing, it needs to be defined for every P, not just the permissions for the class R in which m is declared.

The subtyping relation $\preceq$ must take polymorphism into account. For built-in type, we define $\texttt{bool} \preceq \texttt{bool}$. Class subtyping can be checked using the $\preceq$ function defined in table 2, which propagates instantiation of a class up through the class hierarchy. The definition uses another function, $instance$, that carries out this propagation and constructs a suitable instantiation of a supertype. For use in inference, we need to generate a set of constraints such that two types with variables are in the $\preceq$ relation; this is the purpose of function $tcomp$.

Noninterference. Like FlowCaml [Sim03a], our system uses a level-polymorphic language both for more expressive libraries and because it is the natural result from inference. The noninterference property asserted by a polymorphic type is taken to be ordinary noninterference for all ground instances (satisfying the constraints that are part of the type). In order to show that the constrained types produced by our inference algorithm do imply noninterference, we have defined the type system in this section in which, for commands, only ground instances are typed. This type system is concerned with a closed program, which simplifies the definition of noninterference. The soundness proof in this paper shows the following: For the constrained types inferred for an analysis unit, $unit$, relative to the signature for a library lib, if CT is the classtable obtained as, roughly, the union of $unit$ and lib, then CT is accepted accepted by the type system.

Table 2. Auxiliary functions

These four functions are defined only for T, T' such that $T \leq T'$ in terms of the ordinary class subtyping.

$$\begin{array}{l}
\mathit{instance}(T\langle\overline{\kappa}\rangle, T) = T\langle\overline{\kappa}\rangle \\
\mathit{instance}(T\langle\overline{\kappa}\rangle, T') = \\
\quad \texttt{let}\ CT(T) = \texttt{class}\ T\langle\overline{\alpha}\rangle\ \texttt{extends}\ T''\langle\overline{\lambda}\rangle \\
\quad\ \texttt{in}\ \mathit{instance}(T''\langle\overline{\lambda}[\overline{\alpha} \leftarrow \overline{\kappa}]\rangle, T')
\end{array}$$

$$T\langle\overline{\kappa}\rangle \preceq T'\langle\overline{\kappa'}\rangle \ \Leftrightarrow\ \mathit{instance}(T\langle\overline{\kappa}\rangle, T') = T'\langle\overline{\kappa'}\rangle$$

$$\begin{array}{l}
\mathit{tcomp}(T\langle\overline{\lambda}\rangle, T'\langle\overline{\lambda'}\rangle) = \\
\quad \texttt{let}\ T'\langle\overline{\lambda''}\rangle = \mathit{instance}(T\langle\overline{\lambda}\rangle, T') \\
\quad\ \texttt{in}\ \bigcup_{for\ all\ i}\{\lambda_i'' \leq \lambda_i'; \lambda_i' \leq \lambda_i''\}
\end{array}$$

$$\begin{array}{l}
T \downarrow T = \mathit{true} \\
T \downarrow T' = \\
\quad \texttt{let}\ CT(T') = \texttt{class}\ T'\langle\overline{\alpha}\rangle\ \texttt{extends}\ T''\langle\overline{\lambda}\rangle \\
\quad\quad \texttt{in}\ \overline{\alpha} = \overline{\lambda} \quad \wedge \quad T'' \downarrow T
\end{array}$$

We are confident that the type system ensures termination-insensitive noninterference because it is closely related to the one in [BN03b, BN03a] for which noninterference has been proved. One difference is the rules for test, but it is straightforward to adapt the old proof to the new rules. The more substantial difference is polymorphic classes and methods. We conjecture that any closed unit CT typable by the rules in this section gives rise to a program with no polymorphism: just consider all ground instances, so that $C\langle L\rangle$ and $C\langle H\rangle$ are distinct and incomparable class names. It should not be difficult to check that this is consistent, given the constraints; moreover, it should yield a closed unit CT' that is typable according to [BN03a] — and for which the noninterference property is essentially the same as what one would define for the polymorphic language. A rigorous proof is left to future work.

Security typing rules. Tables 8 and 9 in the appendix give the complete typing rules for expressions and commands. Here we mention only a few cases.

Any downcast of $C\langle\overline{\lambda}\rangle$ to $C'\langle\overline{\lambda}'\rangle$ has $C' \leq C$ as otherwise the program would not by typable as ordinary code. But in addition we must disallow downcasts in the case that information about $C\langle\overline{\lambda}\rangle$ is insufficient to determine soundness of the instantiation $C'\langle\overline{\lambda}'\rangle$. For example, there is the possibility that the declaration of C', say $C'\langle\overline{\alpha}'\rangle$, has more type variables than that for C. Thus, the rule for downcast uses an extra condition, $C \downarrow C'$, also defined in table 2, that essentially make sure that the paramaters for C and C' are the same and appear in the same order through the subclassing chain between them.

Recall that for expressions and commands, the typing rules only apply to ground judgements. The rule for field access is:

$$\frac{(U, \kappa_1) = lsfield(f, W) \qquad U \preceq U' \qquad \Delta \vdash e : W, \kappa_2 \qquad \kappa_2 \leq \kappa_3 \qquad \kappa_1 \leq \kappa_3}{\Delta \vdash e.f : U', \kappa_3}$$

A security typing context Δ is a mapping from variable names to security types. The judgement $\Delta \vdash e : U, \kappa$ says that expression e in context Δ has security type (U, κ). A judgement $\Delta; P \vdash S : \texttt{com}\ \kappa_1, \kappa_2$ says that in the context Δ, if permissions P are excluded then command S writes no variables below κ_1 and no fields below κ_2. Here is the rule for field update.

$$\frac{U, \kappa_1 = lsfield(f, W) \qquad \Delta \vdash e' : U', \kappa_2 \qquad \Delta \vdash e : W, \kappa_3 \qquad U' \preceq U \qquad \kappa_3 \leq \kappa_1 \qquad \kappa_2 \leq \kappa_1 \qquad \kappa_5 \leq \kappa_1}{\Delta; P \vdash e.f := e' : \texttt{com}\ \kappa_4, \kappa_5}$$

The rule precludes the updates of L-fields of a H object reference; otherwise aliasing can be used to leak the H-object [BN03a].

There are two rules for permission testing, as shown below. In the first, if P' is disjoint from P then the test may or may not fail. Hence we need to analyze both branches S_1 and S_2. If P' and P are not disjoint then some subset of P' belongs to the permissions that are excluded. Hence the test *must* fail and only S_2 need be analyzed. The improvement in the rules for permission testing from our earlier work [BN03a, BN03b] is that the static policy is taken into account.

$$\frac{P' \cap P = \emptyset \qquad P' \subseteq Auth(\Delta^{\dagger}\texttt{self}) \qquad \Delta; P \vdash S_1 : \texttt{com}\ \kappa_1, \kappa_2 \qquad \Delta; P \vdash S_2 : \texttt{com}\ \kappa_3, \kappa_4 \qquad \kappa_5 \leq \kappa_1 \sqcap \kappa_3 \qquad \kappa_6 \leq \kappa_2 \sqcap \kappa_4}{\Delta; P \vdash \texttt{test}\ \ P'\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \texttt{com}\ \kappa_5, \kappa_6}$$

$$\frac{P' \cap P \neq \emptyset \vee P' \not\subseteq Auth(\Delta^{\dagger}\texttt{self}) \qquad \Delta; P \vdash S_2 : \texttt{com}\ \kappa_1, \kappa_2 \qquad \kappa_5 \leq \kappa_1 \qquad \kappa_6 \leq \kappa_2}{\Delta; P \vdash \texttt{test}\ \ P'\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \texttt{com}\ \kappa_5, \kappa_6}$$

For method calls the rule uses the polymorphic signature function, and requires that there must be some satisfying ground instance compatible with the levels at the call site.

$$\frac{\begin{array}{c} lsmtype(m, W, P) = \lambda_0, (\overline{U'}, \overline{\lambda}) \text{–} \langle\lambda_1\rangle \rightarrow U', \lambda_2 | K \qquad \Delta, x : (U, \kappa) \vdash e : W, \kappa_3 \qquad \Delta, x : (U, \kappa) \vdash \overline{e} : \overline{U}, \overline{\kappa_4} \\ \kappa_5 \leq \kappa \qquad \kappa_3 \leq \kappa \qquad K' \text{ is satisifable} \\ K' = K \cup \{\overline{\kappa_4} \leq \overline{\lambda}, \lambda_2 \leq \kappa, \kappa_6 \leq \lambda_1, \kappa_3 \leq \lambda_0, \kappa_3 \leq \lambda_1\} \\ \cup tcomp(U', U) \cup tcomp(\overline{U'}, \overline{U}) \end{array}}{\Delta, x : (U, \kappa); P \vdash x := e.m(\overline{e}) : \texttt{com}\ \kappa_5, \kappa_6}$$

The other rules for commands and expressions are in the appendix.

Table 3 gives the rules for class and method declarations. A class declaration is typable provided that all of its method declarations are typable, for all permission sets. To type a method declaration for permission set P requires checking its body with respect to all ground instances of the type given by $lsmtype$ for P and with respect to the relevant permission set $P \cap Auth\ C$.

4 Inference

One input to the inference algorithm is the auxiliary functions giving the signatures of a library. These are named $lsfield$ and $lsmtype$. For proving soundness, we also need the library code, assumed typable with respect to these signatures.

The other input is the analysis unit. We assume all level variables in the analysis unit are distinct, as the objective is to generate constraints on these variables. In order to avoid undecidability related to polymorphic recursion, classes in current analysis unit are treated in a monomorphic way. This is handled using the signature functions, $usfield$ and $usmtype$ for the analysis unit.

Because all variables in the analysis unit are distinct, the corresponding $usmtype$ function does not satisfy invariance with respect to subtyping. Instead, from the result of inference we obtain sufficient constraints with which to construct a method typing function that is invariant. Moreover, we must check that the new classes did not impose additional constraints on methods that are defined in the library —additional constraints would invalidate the assumptions under which the library was analyzed. This is discussed in detail later.

Now we define the signature functions for the class in current unit. The field type function, $usfield$, depends on $ldfield$ since it may inherit fields from the library. $usmtype$, on the other hand, is not compatible with lib method so far because of monomorphism. These new signature functions apply only to classes in current unit. Consequently, the class types in the input are not parameterized.

- $usfield(f, T)$ is defined if $field(f, T)$ is defined. Assume $unit(T) = \texttt{class}\ C\ \texttt{extends}\ U\{\ldots\}$.
  If $U.f$ is undefined, $usfield(f, T)$ is the field type as declared in T.
  If $U.f$ is defined and $U \in unit$, $usfield(f, T) = usfield(f, U)$.
  If $U.f$ is defined and $U \in lib$, $usfield(f, T) = lsfield(f, U)$. In this case, U may have arguments that $lsfield$ instantiates.

- $usmtype(m, T, P)$ is defined if $mtype(m, T)$ is defined.Assume $CT(T) = \texttt{class}\ C\ \texttt{extends}\ U\{\ldots\}$.
  If $U.m$ is undefined, $usmtype(m, T, P)$ is the method type as declared in T.
  If $U.m$ is defined and $U \in unit$, $usmtype(m, T, P) = usmtype(m, U, P)$.
  If $U.m$ is defined and $U \in lib$, $usmtype(m, T, P)$ is the method type as declared in T.

In subsection 4.2 we give an example to illustrate the process of making a parameterized library.

4.1 Inference rules

The inference algorithm is presented in the form of rules for a judgement that generates constraints and keeps track of variables in use (in order to ensure freshness where needed). For expressions, the judgement has the form

$$\Delta; V \vdash e : U \rightsquigarrow \alpha, K, V'$$

Here V, V' are sets of level variables, with $V \subseteq V'$. The judgement means that expression e has level α provided the constraints in K are satisfied. Each rule also has a condition to ensure freshness of new variables, e.g., α is not in V. The constraints K may be expressed using other new variables; V' collects all the new and existing variables. The correctness property is that for any ground instantiation I of V' that satisfies K results in a typable expression (once we instantiate α and the other variables. This is formalized in the soundness theorems.

There is a similar judgement for statements.

Freshness is not required for soundness but it makes the results more general. We do not address completeness in this paper; incompleteness is a consequence of avoiding polymorphic recursion.

The full set of rules is in the Appendix. We discuss just a few cases, which are given in table 4. The first rules in the table are for field access and variable assignment. These may help the reader become familiar with the notation.

The most complicated rules are the two for method invocation. One is for the case of a method that is already defined in the library; the other case is that a method only occurs in (one or more classes in) the analysis unit.

For the case of a method defined in the library, a fresh renaming of the type is needed. For that, I^{*V} is a substitution that maps every variable to a fresh unique variable that does not appear in V. Constraints are generated to enforce the usual conditions relating the method type to the target variable x and the argument expressions $e, \overline{e}$. Moreover, function $tcomp$ is used (from table 2) to generate constraints needed for sound instantiation of parameterized types.

For a method defined in the analysis unit, the rule is similar but there is no need for renaming of variables nor are there constraints given by $usmtype$. Rather, a single set of constraints is being accumulated for the entire unit, including constraints on the method type. Every instance of such a method type has the same variables and thus we avoid the undecidability associated with polymorphic recursion.

Table 3. Typing rule for method and class

$$\frac{\begin{array}{r} lsmtype(m, C\langle\overline{\kappa'}\rangle, P) = \lambda_0, (\overline{U}, \overline{\lambda})\text{–}\langle\lambda_3\rangle\!\rightarrow\! U, \lambda_4 | K \qquad V = vars(\lambda_0, (\overline{U}, \overline{\lambda})\text{–}\langle\lambda_3\rangle\!\rightarrow\! U, \lambda_4 | K) \\ \text{for all } I \text{ with } ok(K, V, I), \;\; \texttt{let}\ \kappa_0, (\overline{U'}, \overline{\kappa})\text{–}\langle\kappa_3\rangle\!\rightarrow\! U', \kappa_4 = I(\lambda_0, (\overline{U}, \overline{\lambda})\text{–}\langle\lambda_3\rangle\!\rightarrow\! U, \lambda_4) \\ \texttt{in}\ \overline{x} : (\overline{U'}, \overline{\kappa}), \texttt{self} : (C, \kappa_0), \texttt{result} : (U', \kappa_4); P \cap Auth(C) \vdash S : \texttt{com}\ \kappa_1, \kappa_2 \qquad \kappa_3 \leq \kappa_2 \end{array}}{C\langle\overline{\kappa'}\rangle\ \texttt{extends}\ R; P \vdash U\ m(\overline{U}\ \overline{x})\{S\}}$$

$$\frac{\text{for all } M \in \overline{M}, \text{ all } \kappa, \text{ and all } P \quad C\langle\overline{\kappa}\rangle\ \texttt{extends}\ I(R); P \vdash I(M) \qquad \text{where } I = [\overline{\alpha} \leftarrow \overline{\kappa}]}{\vdash \texttt{class}\ C\langle\overline{\alpha}\rangle\ \texttt{extends}\ R\{\overline{U}\ \overline{f}; \overline{M}\}}$$

The rules apply by structural recursion to a method body, generating constraints for its primitive commands (like assignment and method call) and constraints for combining these constituents (like in if/else). The first rule in table 5 matches a method body with its declared type and checks it, generating an additional constraint.

The rule for class declaration, also in table 5 combines the constraints for all its methods. This is slightly intricate because we have to check each method declaration $M \in \overline{M}$ with respect to each permission set P (not just $P \subseteq Auth(C)$ for the current class, as method types are indexed on all permission sets). So for n methods and k permissions there are $2^k * n$ method checks to perform. (Substantial optimization can be done by simple analysis of the relevant permissions for a method body and exploiting subsumption properties of security types.)

We refrain from stating a formal rule for the complete analysis unit. The conclusion, written

$$lsmtype, lsfield, usmtype, usfield \vdash unit \rightsquigarrow K, V$$

depends on two hypotheses. First, each class declaration in $unit$ has been checked by the rule in the table, yielding constraints K over variables V. This check is obtained by enumerating the class declarations in $unit$, threading variable sets from one class to the next, and then taking for K the union of the constraints. (Threading is simpler than for methods, because quantification over permission sets is done in the rule for a single class.) The second hypothesis is that overriding declarations do not introduce new constraints, which would invalidate the analysis of the library which is assumed in the form of $lsmtypes$. If this check fails, the analysis fails.

Rather than delving into algorithmic optimizations, we just specify the check for overriding declarations. For each method m that is defined in some class C in the library, and each class D in $unit$ that has an overriding declaration of m, the following must hold. Let

$$lsmtype(m, C\langle\overline{\alpha}\rangle, P) = \lambda_0, (\overline{U}, \overline{\lambda})\text{–}\langle\lambda_1\rangle\!\rightarrow\! W, \lambda_2 | K_0$$
$$usmtype(m, D\langle\overline{\beta}\rangle, P) = \lambda'_0, (\overline{U'}, \overline{\lambda'})\text{–}\langle\lambda'_1\rangle\!\rightarrow\! W', \lambda'_2$$

and let K be the constraints generated as above for $unit$. Let K_0^* be the closure of K_0 under implication, and K^* the closure of K under implication. Let $rest(K_0^*, m, C, P)$ be the restriction of K_0^* to variables that occur in $lsmtype(m, C\langle\overline{\alpha}\rangle, P)$, and likewise $rest(K^*, m, D, P)$ be the restriction for $usmtype$. Then $rest(K^*, m, D, P)$ should be implied by $rest(K_0^*, m, C, P)$.

The point is that we only compare constraints for a particular method —it is certainly not the case that the constraints from the library imply all constraints for $unit$, e.g., the unit can have additional methods. On the other hand, it is unsound to simply restrict K and K_0 to the relevant variables, as this could omit implied constraints.

4.2 Building a new library

In this subsection we illustrate the manipulation of parametized classes, resulting in a new library signature. Then we give the definitions. Besides practical use, this is needed to formulate soundness.

Example. Assume we have a input program, in which we have already put variables where we want to infer, and constants where we want to fix the levels. Suppose $A\langle\alpha\rangle$ is defined in the library. Note that we have recursive references between the two classes, which is the reason why they have to be put in the same unit for analysis. We only show the types of the fields and method types here.

class B **extends** Object{
(B,α_1) m1();
(B,α_2) m2();
x:(A<α_3>,H);
y:(D,α_4); }

Table 4. Inference algorithm: selected expression and statement cases.

$$\frac{\Delta, V \vdash e : W \rightsquigarrow \lambda_2, K_1, V_1 \quad (U,\lambda_1) = lsfield(f,W) \vee (U,\lambda_1) = usfield(f,W) \quad K' = K_1 \cup \{\lambda_1 \le \alpha_3, \lambda_2 \le \alpha_3\}, \{\alpha_3\} \cup V_1 \quad \alpha_3 \notin V_1}{\Delta, V \vdash e.f : U \rightsquigarrow \alpha_3, K', \{\alpha_3\} \cup V_1}$$

$$\frac{\Delta, x : (U,\lambda_1) \vdash e : U' \rightsquigarrow \lambda_2, K_1, V_1 \quad K' = K_1 \cup \{\lambda_2 \le \lambda_1, \alpha_3 \le \lambda_1\} \cup tcomp(U',U) \quad \alpha_3, \alpha_4 \notin V_1}{P, \Delta, x : (U,\lambda_1), V \vdash x := e; \rightsquigarrow \texttt{com}\ (\alpha_3,\alpha_4), K', \{\alpha_3,\alpha_4\} \cup V_1}$$

$$\frac{\begin{array}{c}\Delta, x : (U,\lambda), V \vdash e : W \rightsquigarrow \lambda_3, K_0, V_0 \\ \Delta, x : (U,\lambda), V_0 \vdash \overline{e} : \overline{U} \rightsquigarrow \lambda_4, K_1, V_1 \quad I^{*V_1}(lsmtype(m,W,P)) = (\lambda_0, (\overline{U'}, \overline{\lambda}) \multimap \langle\lambda_1\rangle \rightarrowtail R, \lambda_2) | K \\ V' = V_1 \cup var(K) \cup var(R) \cup var(\overline{U'}) \cup var(\lambda_0, \overline{\lambda}, \lambda_1, \lambda_2) \\ K' = K \cup K_0 \cup K_1 \cup tcomp(\overline{U}, \overline{U'}) \cup tcomp(R,U) \cup \{\overline{\lambda_4} \le \overline{\lambda}, \lambda_3 \le \lambda_0, \lambda_3 \le \lambda_1, \lambda_2 \le \lambda, \alpha_5 \le \lambda, \alpha_6 \le \lambda_1, \lambda_3 \le \lambda\} \\ \alpha_5, \alpha_6 \notin V'\end{array}}{P, \Delta, x : (U,\lambda), V \vdash x := e.m(\overline{e}) \rightsquigarrow \texttt{com}\ (\alpha_5,\alpha_6), K', \{\alpha_5,\alpha_6\} \cup V'}$$

$$\frac{\begin{array}{c}\Delta, x : (U,\lambda), V \vdash e : W \rightsquigarrow \lambda_3, K_0, V_0 \\ \Delta, x : (U,\lambda), V_0 \vdash \overline{e} : \overline{U} \rightsquigarrow \lambda_4, K_1, V_1 \quad usmtype(m,W,P) = (\lambda_0, (\overline{U'}, \overline{\lambda}) \multimap \langle\lambda_1\rangle \rightarrowtail R, \lambda_2) \\ K' = K_0 \cup K_1 \cup tcomp(\overline{U}, \overline{U'}) \cup tcomp(R,U) \cup \{\overline{\lambda_4} \le \overline{\lambda}, \lambda_3 \le \lambda_0, \lambda_3 \le \lambda_1, \lambda_2 \le \lambda, \alpha_5 \le \lambda, \alpha_6 \le \lambda_1, \lambda_3 \le \lambda\} \\ \alpha_5, \alpha_6 \notin V_1\end{array}}{P, \Delta, x : (U,\lambda), V \vdash x := e.m(\overline{e}) \rightsquigarrow \texttt{com}\ (\alpha_5,\alpha_6), K', \{\alpha_5,\alpha_6\} \cup V_1}$$

class D **extends** Object{
z:(B,α_5); }

We run the algorithm, and have the result $\{\alpha_1,\alpha_2,\alpha_3,\alpha_4,\alpha_5\}|K$ for some K. To put the program into the library, we must put it into a parameterized format. First, we collect all the variables appearing in field declaration for each class. In this example, it happen to be $\{\alpha_3,\alpha_4,\alpha_5\}$ for both B and D. This will be used as the formal parameter list for the classes. Then we convert all type references to the class defined in the current unit to its polymorphic form by appending the list of parameters exactly the same as the formal parameters. For this case, all B becomes $B<\alpha_3,\alpha_4,\alpha_5>$, and all D becomes $D<\alpha_3,\alpha_4,\alpha_5>$. As for constraint for the method, we need to attach K to every method. So after the conversion, the program signature now becomes:

class B<$\alpha_3,\alpha_4,\alpha_5$> **extends** Object{
(B<$\alpha_3,\alpha_4,\alpha_5$>,$\alpha_1$) m1() | K
(B<$\alpha_3,\alpha_4,\alpha_5$>,$\alpha_2$) m2() | K
x:(A<α_3>,H);
y:(D<$\alpha_3,\alpha_4,\alpha_5$>,$\alpha_4$); }
class D<$\alpha_3,\alpha_4,\alpha_5$> **extends** Object{
z:(B<$\alpha_3,\alpha_4,\alpha_5$>,$\alpha_5$); }

Converting. By the algorithm we can get (K,V) on the classes in the unit. To type-check it, or to prepare it for later use as part of the library, we need to do some transformation.

For converting the code, let X be the set of class names declared in $unit$. We study any class $C \in X$. Let V' be all the variables in V that appear in the supertype or field type/label of C. Let $unit'$ be $unit$ but with every C in X replaced by $C<V'>$. The $unit'$ is now a parameterized class with polymorphic methods.

Now we need to combine the signatures from the library and the unit. Based on $unit'$, we will build new signature function that can access the converted unit and the library uniformly.

$fieldmerge(lsfield, usfield)(f, T<\overline{\lambda}>) =$
  `if` $T \in lib$
    $lsfield(f, T<\overline{\lambda}>)$
  `else`
    `let` $unit(T) =$ `class` $T<\overline{\alpha}> \ldots$ `in`
    $usfield(f,T)[\overline{\alpha} \leftarrow \overline{\lambda}]$

$methmerge(lsmtype, usmtype, K)(m, T<\overline{\lambda}>, P) =$
  `if` m is initially defined in class D $\in lib$
    $lsmtype(m, \textit{instance}(T<\overline{\lambda}>, D), P)$
  `else`
    `let` $unit(T) =$ `class` $T<\overline{\alpha}> \ldots$ `in`
    $((usmtype(m,T,P)|K)[\overline{\alpha} \leftarrow \overline{\lambda}]$

Table 5. Inference algorithm for method, class and unit

$$\frac{usmtype(m,C,P) = \lambda_0, (\overline{U},\overline{\lambda})\text{–}\langle\lambda_3\rangle\!\rightarrow\! R, \lambda_4 \quad \Delta = [\overline{x} : (\overline{U},\overline{\lambda}); \texttt{self} : (U,\lambda_0);\ \texttt{result} : (R,\lambda_4)] \qquad P,\Delta,V \vdash S \rightsquigarrow \texttt{com}\,(\alpha_1,\alpha_2), K_1, V_1}{P, C\ \texttt{extends}\ D, V \vdash R\ m(\overline{U}\ \overline{x})\{S\} \rightsquigarrow K_1 \cup \{\lambda_3 \le \alpha_2\}, V_1}$$

$$\frac{\begin{array}{c}\text{let } (P_0,M_0),\ldots(P_n,M_n) \text{ be an enumeration of all pairs with each } M_i \in \overline{M} \text{ and } P_i \subseteq Permissions \text{ in} \\ \text{forall } i \quad P_i, C\ \texttt{extends}\ D, V_{i-1} \vdash M_i \rightsquigarrow K_i, V_i\end{array}}{V_0 \vdash \texttt{class}\ C\ \texttt{extends}\ D\{\overline{U}\ \overline{f}, \overline{M}\} \rightsquigarrow \bigcup_{1\le i\le n} K_i, V_n}$$

4.3 Complexity analysis

Assumption:

- There is a very good hash function, such that adding two set takes linear time.
- Table lookup, with the help of the above hash function, take constant time.

The performance of the system depends on several issue. First, we consider the inference algorithm. For each method, the algorithm traverses through the method body inductively. So not considering library method calls, the number of constraints is proportional to the length n of the unit. Assume all library constraints have been simplified. Each lib class has a parameter list of length t. Each lib method has a signature of size s. Worst time cost for inference of a method in current unit will be $O(n(s+t)^2)$, in a slightly different version of the algorithm where the operation of union has been optimized.

Considering the permissions, the size of the constraint set is $O(n(s+t)^2 2^P)$. As discussed at the end of section 4.1, we need to check if the methods overrides library signatures properly. The same algorithm can be used to simplify the scheme (K, V) with respect to a given method signature $V' \subseteq V$.The time cost is $O(|K||V'|)$. In our case, checking all types in the unit takes $O(mn(s+t)^3 2^{2|P|})$ time, assuming the number of method is m.

Satisfiability. We can use an algorithm [RM99] that checks the satisfiability of the constraint set in $O(|K|)$ time. So the check in our case takes $O(n(s+t)^2 2^{|P|})$.

Space cost is much smaller. Assume all constraints set has been simplified with respect to each individual method. Each method has a signature of size $O((s+t)^2)$, which is the square of the variables it may contains. So the signature of the unit is of size $O(m(s+t)^2 2^{|P|})$.

5 Soundness

Theorem 1 (Soundness) Suppose $unit'$ is the converted $unit$,
$sfield = fieldmerge(lsfield, usfield)$,
and $smtype = methmerge(lsmtype, usmtype, K)$. If

$$lsfield, lsmtype, usfield, usmtype \vdash unit \rightsquigarrow K, V$$

and every declaration lib is typable then $sfield, smtype \vdash unit'$.

$Proof$: The conclusion, $sfield, smtype \vdash unit'$, typability of $unit'$, means that every well-formed class declaration

$$\texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{extends}\ D\texttt{<}\overline{\lambda}\texttt{>}\{(\overline{U},\overline{\lambda_1})\ \overline{f};\overline{M};\}$$

in $unit'$ is typable. For declarations in $unit'$ that come from lib this is by hypothesis. The rest of the proof concerns declarations from $unit$.

According to the typing rule in Table 3, we have to check, for every $P \subseteq Permissions$, every $I : \overline{\alpha} \rightarrow levels$ and every $M \in \overline{M}$, that

$$sfield, smtype, C\texttt{<}\overline{I(\alpha)}\texttt{>}\ \texttt{extends}\ D\texttt{<}\overline{I(\lambda)}\texttt{>}, P \vdash I(M)$$

The rule for checking this judgement (first rule in table 3) says: let $\lambda_0, (\overline{U},\overline{\lambda})\text{–}\langle\lambda_3\rangle\!\rightarrow\! U, \lambda_4 | K' = lsmtype(m, C, P)$ where $FV(K') \subseteq V$. Let

$$\Delta = [\overline{x} : I(\overline{U},\overline{\lambda}),\ \texttt{result} : I(U,\lambda_4), \texttt{self} : I(C\texttt{<}\overline{\alpha}\texttt{>},\lambda_0)]$$

For any $J : V \rightarrow levels$, such that $J \supseteq I \wedge ok(K', V, J)$, we have to show

$$J(\Delta), P \vdash J(S') : \texttt{com}\ J(\alpha_1), J(\alpha_2) \qquad (*)$$

where S' is the body of the method in $unit'$, i.e., with type parameters added. We also have to show $J(\lambda_3) \le J(\alpha_2)$; it holds because $(\lambda_3 \le \alpha_2) \in K_m$.

By hypothesis of the theorem,

$$lsfield, lsmtype, usfield, usmtype \vdash unit \rightsquigarrow K, V$$

By the definition of the analysis algorithm, this means the class was analyzed, which means that each method was analyzed and their constraints were unioned together. Thus there exists K_m,V_m such that $K_m \subseteq K$ and $V_m \subseteq V$ and

$$P, J(\Delta), typesigs \vdash S' \rightsquigarrow \texttt{com}\,(J(\lambda_1), J(\lambda_2)), K_m, V_m$$

where $typesigs$ is $lsfield, usfield, lsmtype, usmtype$ (for reasons of typesetting) and S' is the method body in $unit$, i.e., without type parameters. Now $(*)$ follows, using lemma 1, Q.E.D.

We continue to write $typesigs$ for $lsfield$,$usfield$, $lsmtype$, $usmtype$ and K_u, V_u for the constraints and variables for $unit$. Moreover,
$sfield = fieldmerge(lsfield, usfield)$
$smtype = methmerge(lsmtype, usmtype, K_u)$.

Lemma 1 (Soundness for command) Suppose

$$typesigs, P, \Delta, V \vdash S' \rightsquigarrow \texttt{com}\,(\lambda_1, \lambda_2), K', V'$$

$K' \subseteq K_u$, $V' \subseteq V_u$, and $ok(K_u, V_u, I)$. Then

$$sfield, smtype, I(\Delta), P \vdash I(S) : \texttt{com}\, I(\lambda_1), I(\lambda_2)$$

Proof:
Field assignment case:
Suppose

$$P, \Delta, V \vdash e.f := e'; \rightsquigarrow \texttt{com}\,(\alpha_4, \alpha_5), K', V'$$

By the structure of the algorithm, we know that: $K' = K_1 \cup K_2 \cup tcomp(U', U) \cup \{\lambda_2 \leq \lambda_1; \lambda_3 \leq \lambda_1; \alpha_5 \leq \lambda_1\}$ and $V' = \{\alpha_4, \alpha_5\} \cup V_2$ where

$$\Delta, V \vdash e' : U' \rightsquigarrow \lambda_2, K_1, V_1$$
$$\Delta, V_1 \vdash e : T\langle\overline{\lambda}\rangle \rightsquigarrow \lambda_3, K_2, V_2$$
$$(U, \lambda_1) = lsfield(f, T\langle\overline{\lambda}\rangle) \vee (U, \lambda_1) = usfield(f, T)$$

If $U \in lib$, by definition, $sfield(f, T\langle\overline{\lambda}\rangle) = lsfield(f, T\langle\overline{\lambda}\rangle) = (U, \lambda_1)$. If $U \in unit'$, because of the monomorphsim within the unit and the correspondence between S and S', the actual parameter of T are always exactly the same as the formal parameter declared in the class template of T. Then $sfield(f, T\langle\overline{\lambda}\rangle) = usfield(f, T)[\overline{\lambda} \leftarrow \overline{\lambda}] = usfield(f, T) = (U, \lambda_1)$. So in both cases, $(U, \lambda_1) = sfield(f, W)$. Because $ok(K', V', I)$ we can have $I(K_1), I(K_2), I(\lambda_2) \leq I(\lambda_1), I(\lambda_3) \leq I(\lambda_1), I(\alpha_5) \leq I(\lambda_1), I(tcomp(U', U))$. Following the definition of $tcomp$, we can get $tcomp(I(U'), I(U))$.

Now we have $I(\Delta); P \vdash I(e') : I(U'), I(\lambda_2)$ and $I(\Delta); P \vdash I(e) : I(T\langle\overline{\lambda}\rangle), I(\lambda_3)$ by the soundness lemma for expressions. Moreover, $sfield(f, I(T\langle\overline{\lambda}\rangle)) = I(U), I(\lambda_1)$ and $I(U') \preceq I(U)$ because $tcomp$ generates constraints to ensure this. One can also check $I(\lambda_2) \leq I(\lambda_1)$, $I(\lambda_3) \leq I(\lambda_1)$, and $I(\alpha_5) \leq I(\lambda_1)$, which complete the hypotheses of the typing rule. Thus the rule yields

$$I(\Delta); P \vdash I(e.f := e') : \texttt{com}\,(I(\alpha_4), I(\alpha_5))$$

and the proof for field access is complete.

The intra-unit method call case:
There are two type of method call: library call or call on a class in the current unit. We only prove the latter, which is more interesting.

$$P, \Delta, x : (U, \lambda) \vdash x := e.m(\overline{e}) \rightsquigarrow \texttt{com}\,(\alpha_5, \alpha_6), K', V'$$

By the structure of the algorithm, we know

$$\Delta, x : (U, \lambda), V \vdash e : T\langle\overline{\lambda'}\rangle \rightsquigarrow \lambda_3, K_0, V_0$$
$$\Delta, x : (U, \lambda), V_0 \vdash \overline{e} : \overline{U} \rightsquigarrow \lambda, K_1, V_1$$

Let

$$(\lambda_0, (\overline{U'}, \overline{\lambda}) \text{–} \langle\lambda_1\rangle \rightarrow R, \lambda_2) | K_t) = smtype(m, T\langle\overline{\lambda'}\rangle, P)$$

Since $T \in unit'$, by the definition of $methmerge$,

$$(\lambda_0, (\overline{U'}, \overline{\lambda}) \text{–} \langle\lambda_1\rangle \rightarrow R, \lambda_2) | K_t = (usmtype(m, T, P) | K_u)[\overline{\lambda'} \leftarrow \overline{\lambda'}] = usmtype(m, T, P) | K_u$$

where the substitution can be removed because of intra-unit monomorphsim. This implies $K_t = K_u$. By assumption, $I(K_u)$ holds. In addition, we have

$$K_u \supseteq K' \supseteq tcomp(\overline{U'}, \overline{U}) \cup tcomp(R, U) \cup \{\lambda_4 \leq \lambda, \lambda_3 \leq \lambda_0, \lambda_3 \leq \lambda_1, \lambda_2 \leq \lambda, \alpha_6 \leq \lambda_1\} \cup K_1 \cup \{\alpha_5 \leq \lambda, \lambda_3 \leq \lambda\}$$

Furthermore, we get

$$P, I(\Delta), x : (I(U), I(\lambda)) \vdash e : I(T\langle\overline{\lambda'}\rangle), I(\lambda_3)$$
$$P, I(\Delta), x : (I(U), I(\lambda)) \vdash \overline{e} : I(\overline{U}), I(\lambda_4)$$

by the soundness lemma for expressions. One can check that by ok for I we have the required conditions $I(\alpha_5) \leq I(\lambda)$ and $I(\lambda_3) \leq I(\lambda)$. Moreover, let $smtype(m, T\langle\overline{\lambda'}\rangle, P) = (\lambda_0, (\overline{U'}, \overline{\lambda}) \text{–} \langle\lambda_1\rangle \rightarrow R, \lambda_2) | K_t$ and let $K = K_t \cup tcomp(\overline{U'}, \overline{U}) \cup tcomp(R, U) \cup \{\lambda_3 \leq \lambda_0, \lambda_3 \leq \lambda_1, \lambda_2 \leq \lambda, \alpha_6 \leq \lambda_1\}$; then $I(K)$ holds. Thus by the typing rule we get

$$P, I(\Delta), x : (I(U), I(\lambda)) \vdash I(x := e.m(\overline{e})) : \texttt{com}\,(I(\alpha_5), I(\alpha_6))$$

Q.E.D.

The soundness lemma for expressions is similar to that for commands, and is omitted.

6 Related Work

Volpano and Smith [VS97], building on their seminal work [VSI96], give a security type system and an inference algorithm for a simple procedural language. The type system guarantees noninterference: a well-typed program does not leak sensitive data. Moreover, they provide an inference algorithm that infers security types. To do this, they add level variables to the security type system and the inference algorithm infers security types together with constraints between variables of the security types. They prove the algorithm sound and complete with respect to the type system. Their type system has principal types (the types appear together with constraints) and the inference algorithm computes the principal types. However, they do not handle libraries.

Myers [Mye99a, Mye99b] gave a security typing system for full Java but left open the problem of justifying the rules with a noninterference result. This is not surprising, as the language is substantial and the rules are complex. Myers, Zdancewic and their students have implemented a security model of the Java language [Mye99b, ML97], Jif. The Jif compiler[1] handles several advanced features like exceptions, declassification, dynamic labels and polymorphism. However, it appears that inference in the system is restricted to the labels of local variables and loop invariants within the method body. Field and method types are added either manually or by default.

Simonet recently presented a version of ML with security flow labels, termed FlowCaml[Sim03b, Sim03a]. FlowCaml is based on Objective Caml, and supports polymorphism, exceptions, structural subtyping and the module system. The correctness of the type system and inference system are formally stated and verified [PS02, Sim03a]. It is equipped with a dedicated inference engine[Sim03c]. FlowCaml handles pointers and mutable state but does not handle typical features of a class-based object oriented languages, e.g., extended state in the form of subclassing, method inheritance and override.

Banerjee and Naumann [BN02] present a security type system for secure information flow and prove noninterference for a sequential object-oriented language with pointers and mutable states, private fields, dynamic binding and inheritance, recursive classes and methods. The main contribution of their later work [BN03b, BN03a] is to account for secure information flow in the presence of dynamic access control. The particular access control mechanism studied is stack inspection and the sequential object-oriented language is extended with primitives for access control. A security type system is given and proved to ensure noninterference. But they do not handle security type inference, hence security labels for variables and at method interfaces must be added manually. In ongoing work, Naumann has encoded most of the semantic model and static analysis of [BN03b] and is machine checking the proof of noninterference.

We have a working prototype for a subset of our current language. More specifically, the prototype is for the sequential object-oriented language described in [BN02], i.e., without access control. It accepts a class declaration that is partly annotated with level constants, generating a constraint set and checking its satisfiability. If the code is typable, the output will be a polymorphic type for the given program in its most general form. The inference is nonmodular in that it does not handle libraries but infers types for complete programs.

Our prototype also includes a simplifier that can transform the set of inferred constraints into an equivalent, smaller set. For lack of space, we do not discuss constraint simplification in this paper. A sample output of our prototype for the *IRS* example in Section 2 is shown in the Appendix. The first author's dissertation will include a full implementation of this paper and additional aspects such as constraint minimization.

[1] On the web at `http://www.cs.cornell.edu/jif/`

7 Conclusion

The main contribution of this paper is the specification of a modular algorithm that infers security types in the presence of dynamic access control for a sequential, class-based, object-oriented language. This requires the addition of security level variables to the language and moreover, requires classes parameterized with security levels. The inference algorithm constructs a library where each class is parameterized by the levels in its fields. Each method of a parameterized class can be given a polymorphic, constrained signature. This has the additional benefit of being more expressive and flexible for the programmer. We have shown soundness for the algorithm and work is in progress on a prototype.

References

[Aba99] Martin Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.

[BN02] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–270. IEEE Computer Society Press, 2002.

[BN03a] Anindya Banerjee and David A. Naumann. Stack-based access control for secure information flow. Submitted., 2003.

[BN03b] Anindya Banerjee and David A. Naumann. Using access control for secure information flow in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 155–169. IEEE Computer Society Press, 2003.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[IPW01] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–459, May 2001.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

[Myc84] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Sixth International Symposium on Programming*, number 166 in Lecture Notes in Computer Science. Springer-Verlag, 1984.

[Mye99a] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[Mye99b] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Laboratory of Computer Science, MIT, 1999.

[PS02] François Pottier and Vincent Simonet. Information flow inference for ML. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[RM99] Jakob Rehof and Torben Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35:191–221, 1999.

[Sim03a] Vincent Simonet. Flow Caml in a nutshell. In Graham Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, March 2003.

[Sim03b] Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

[Sim03c] Vincent Simonet. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In *Asian Symposium on Programming Languages and Systems (APLAS'03)*. Springer-Verlag, September 2003.

[VS97] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97*, number 1214 in Lecture Notes in Computer Science, pages 607–621. Springer-Verlag, 1997.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

A Appendix

Output of the prototype on IRS.java

```
The raw scheme before simplification
-------------
class IRS extends Object{
    int income H;
    int k_3 tax k_1,k_2 (int k_4);
}

constraints:
k_4<=k_9, k_6<=k_8, k_12<=k_18, k_10<=H,
k_16<=k_15, k_5<=k_7, k_11<=k_17, k_2<=k_6,
k_12<=k_14, k_9<=H, k_8<=H, k_11<=k_13,
k_1<=k_16, H<=k_15, k_6<=k_12, k_5<=k_11,
 k_1<=k_10, k_15<=k_3, k_13<=k_3;

alphabet:
k_16, k_15, k_14, k_13, k_12, k_11, k_10,
k_9, k_8, k_7, k_6, k_5, k_4, k_3, H, k_2,
k_1, k_18, k_17;

The scheme after simplification
-----------
class IRS extends Object{
    int income H;
    int H tax k_1,k_2 (int k_4);
}

constraints:
;

alphabet:
k_4, H, k_2, k_1;
```

Additional rules

Tables 6 and 7 give typing rules omitted from the main body of the paper.

Tables 8 and 9 give inference rules omitted from the main body.

Table 6. Typing rule for expressions

$$\Delta \vdash \texttt{true} : \texttt{bool}, \kappa \qquad \Delta \vdash \texttt{null} : U, \kappa$$

$$\frac{\Delta \vdash e_1 : U_1, \kappa_1 \qquad \Delta \vdash e_2 : U_2, \kappa_2 \qquad \kappa_1 \sqcup \kappa_2 \leq \kappa_3}{\Delta \vdash e_1 == e_2 : \texttt{bool}, \kappa_3}$$

$$\frac{(U, \kappa_1) = lsdfield(f, W) \qquad U \preceq U' \qquad \Delta \vdash e : W, \kappa_2 \qquad \kappa_2 \leq \kappa_3 \qquad \kappa_1 \leq \kappa_3}{\Delta \vdash e.f : U', \kappa_3}$$

$$\frac{\Delta \vdash e : U', \kappa \qquad U' \preceq U \qquad \kappa \leq \kappa'}{\Delta \vdash (U)e : U, \kappa'}$$

$$\frac{\Delta \vdash e : U', \kappa \qquad U \preceq U' \qquad U' \downarrow U \qquad \kappa \leq \kappa'}{\Delta \vdash (U)e : U, \kappa'}$$

$$\frac{\Delta \vdash e : U', \kappa \qquad U' \preceq U \qquad \kappa \leq \kappa'}{\Delta \vdash e \texttt{ is } U : \texttt{bool}, \kappa'}$$

$$\frac{\Delta \vdash e : U', \kappa \qquad U \preceq U' \qquad U' \downarrow U \qquad \kappa \leq \kappa'}{\Delta \vdash e \texttt{ is } U : \texttt{bool}, \kappa'}$$

Table 7. Typing rules for command

$$\frac{\Delta, x:(U,\kappa_1) \vdash e : U', \kappa_2 \quad U' \preceq U \quad x \neq \texttt{self} \quad \kappa_2 \leq \kappa_1 \quad \kappa_3 \leq \kappa_1}{\Delta, x:(U,\kappa_1); P \vdash x := e : \texttt{com}\,\kappa_3, \kappa_4}$$

$$\frac{U, \kappa_1 = lsdfield(f, W) \quad \Delta \vdash e' : U', \kappa_2 \quad \Delta \vdash e : W, \kappa_3 \quad U' \preceq U \quad \kappa_3 \leq \kappa_1 \quad \kappa_2 \leq \kappa_1 \quad \kappa_5 \leq \kappa_1}{\Delta; P \vdash e.f := e' : \texttt{com}\,\kappa_4, \kappa_5}$$

$$\frac{x \neq \texttt{self} \quad U' \preceq U \quad \kappa_1 \leq \kappa}{\Delta, x:(U,\kappa); P \vdash x := \texttt{new}\, U'() : \texttt{com}\,\kappa_1, \kappa_2}$$

$$\frac{lsmtype(m, W, P) = \lambda_0, (\overline{U'}, \overline{\lambda}) \text{-} \langle \lambda_1 \rangle \rightarrow U', \lambda_2 | C \quad \Delta, x:(U,\kappa) \vdash e : W, \kappa_3 \quad \Delta, x:(U,\kappa) \vdash \overline{e} : \overline{U}, \overline{\kappa_4} \quad \kappa_5 \leq \kappa \quad x \neq \texttt{self} \quad \kappa_3 \leq \kappa \quad K' = K \cup \{\overline{\kappa_4} \leq \overline{\lambda}, \lambda_2 \leq \kappa, \kappa_6 \leq \lambda_1, \kappa_3 \leq \lambda_0, \kappa_3 \leq \lambda_1\} \cup tcomp(U', U) \cup tcomp(\overline{U'}, \overline{U}) \quad K' \text{ is satisifable.}}{\Delta, x:(U,\kappa); P \vdash x := e.m(\overline{e}) : \texttt{com}\,\kappa_5, \kappa_6}$$

$$\frac{\Delta; P \vdash S_1 : \texttt{com}\,\kappa_1, \kappa_2 \quad \Delta; P \vdash S_2 : \texttt{com}\,\kappa_3, \kappa_4 \quad \kappa_5 \leq \kappa_1 \sqcap \kappa_3 \quad \kappa_6 \leq \kappa_2 \sqcap \kappa_4}{\Delta; P \vdash S_1; S_2 : \texttt{com}\,\kappa_5, \kappa_6}$$

$$\frac{\Delta; P \vdash S_1 : \texttt{com}\,\kappa_1, \kappa_2 \quad \Delta; P \vdash S_2 : \texttt{com}\,\kappa_3, \kappa_4 \quad \kappa_5 \leq \kappa_1 \sqcap \kappa_3 \quad \kappa_6 \leq \kappa_2 \sqcap \kappa_4}{\Delta; P \vdash S_1; S_2 : \texttt{com}\,\kappa_5, \kappa_6}$$

$$\frac{\Delta; P \vdash e : \texttt{bool}, \kappa \quad \Delta; P \vdash S_1 : \texttt{com}\,\kappa_1, \kappa_2 \quad \Delta; P \vdash S_2 : \texttt{com}\,\kappa_3, \kappa_4 \quad \kappa \leq \kappa_1 \sqcap \kappa_2 \sqcap \kappa_3 \sqcap \kappa_4 \quad \kappa_5 \leq \kappa_1 \sqcap \kappa_3 \quad \kappa_6 \leq \kappa_2 \sqcap \kappa_4}{\Delta; P \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi} : \texttt{com}\,\kappa_5, \kappa_6}$$

$$\frac{\Delta, x:(U,\kappa_2); P \vdash S : \texttt{com}\,\kappa_3, \kappa_4 \quad \Delta; P \vdash e : U', \kappa_1 \quad U' \preceq U \quad \kappa_1 \leq \kappa_2 \quad \kappa_5 \leq \kappa_3 \quad \kappa_6 \leq \kappa_4}{\Delta; P \vdash U\ x := e\ \texttt{in}\ S : \texttt{com}\,\kappa_5, \kappa_6}$$

$$\frac{P' \cap P = \emptyset \quad P' \subseteq Auth(\Delta^{\dagger}\texttt{self}) \quad \Delta; P \vdash S_1 : \texttt{com}\,\kappa_1, \kappa_2 \quad \Delta; P \vdash S_2 : \texttt{com}\,\kappa_3, \kappa_4 \quad \kappa_5 \leq \kappa_1 \sqcap \kappa_3 \quad \kappa_6 \leq \kappa_2 \sqcap \kappa_4}{\Delta; P \vdash \texttt{test}\ P'\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \texttt{com}\,\kappa_5, \kappa_6}$$

$$\frac{P' \cap P \neq \emptyset \vee P' \not\subseteq Auth(\Delta^{\dagger}\texttt{self}) \quad \Delta; P \vdash S_2 : \texttt{com}\,\kappa_1, \kappa_2 \quad \kappa_5 \leq \kappa_1 \quad \kappa_6 \leq \kappa_2}{\Delta; P \vdash \texttt{test}\ P'\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \texttt{com}\,\kappa_5, \kappa_6}$$

Table 8. Expression inference

$$\Delta; V \vdash \texttt{true} : \texttt{bool} \rightsquigarrow \alpha, \emptyset, \{\alpha\} \cup V \quad \alpha \notin V$$

$$\Delta, x:(U,\lambda); V \vdash x : U \rightsquigarrow \alpha, \{\lambda \leq \alpha\}, \{\alpha\} \cup V \quad \alpha \notin V$$

$$\frac{\Delta, V \vdash e_1 : U_1 \rightsquigarrow \lambda_1, K_1, V_1 \quad \Delta, V_1 \vdash e_2 : U_2 \rightsquigarrow \lambda_2, K_2, V_2 \quad K' = \{\lambda_1 \leq \alpha_3; \lambda_2 \leq \alpha_3\} \quad \alpha_3 \notin V_2}{\Delta, V \vdash e_1 == e_2 : \texttt{bool} \rightsquigarrow \alpha_3, K', \{\alpha_3\} \cup V_2}$$

$$\frac{\Delta, V \vdash e : W \rightsquigarrow \lambda_2, K_1, V_1 \quad (U, \lambda_1) = lsdfield(f, W) \vee (U, \lambda_1) = usdfield(f, W) \quad K' = K_1 \cup \{\lambda_1 \leq \alpha_3, \lambda_2 \leq \alpha_3\}, \{\alpha_3\} \cup V_1 \quad \alpha_3 \notin V_1}{\Delta, V \vdash e.f : U \rightsquigarrow \alpha_3, K', \{\alpha_3\} \cup V_1}$$

$$\frac{T' \preceq T \quad \Delta, V \vdash e : T' \text{<} \overline{\lambda'} \text{>} \rightsquigarrow \lambda, K_1, V_1 \quad K' = K_1 \cup \{\alpha' \leq \lambda\} \cup tcomp(T' \text{<} \overline{\lambda'} \text{>}, T \text{<} \overline{\lambda} \text{>}) \quad \alpha' \notin V_1}{\Delta, V \vdash (T \text{<} \overline{\lambda} \text{>}) e : T \text{<} \overline{\lambda} \text{>} \rightsquigarrow \alpha', K', \{\alpha'\} \cup V_1}$$

$$\frac{T \preceq T' \quad T' \downarrow T \quad \Delta, V \vdash e : T' \text{<} \overline{\lambda'} \text{>} \rightsquigarrow \lambda, K_1, V_1 \quad K' = K_1 \cup \{\alpha' \leq \lambda\} \cup tcomp(T \text{<} \overline{\lambda} \text{>}, T' \text{<} \overline{\lambda'} \text{>}) \quad \alpha' \notin V_1}{\Delta, V \vdash (T \text{<} \overline{\lambda} \text{>}) e : T \text{<} \overline{\lambda} \text{>} \rightsquigarrow \alpha', K', \{\alpha'\} \cup V_1}$$

$$\frac{T' \preceq T \quad \Delta, V \vdash e : T' \text{<} \overline{\lambda'} \text{>} \rightsquigarrow \lambda, K_1, V_1 \quad K' = K_1 \cup \{\alpha' \leq \lambda\} \cup tcomp(T' \text{<} \overline{\lambda'} \text{>}, T \text{<} \overline{\lambda} \text{>}) \quad \alpha' \notin V_1}{\Delta, V \vdash e\ \texttt{is}\ T \text{<} \overline{\lambda} \text{>} \rightsquigarrow \alpha', K', \{\alpha'\} \cup V_1}$$

$$\frac{T \preceq T' \quad T' \downarrow T \quad \Delta, V \vdash e : T' \text{<} \overline{\lambda'} \text{>} \rightsquigarrow \lambda, K_1, V_1 \quad K' = K_1 \cup \{\alpha' \leq \lambda\} \cup tcomp(T \text{<} \overline{\lambda} \text{>}, T' \text{<} \overline{\lambda'} \text{>}) \quad \alpha' \notin V_1}{\Delta, V \vdash e\ \texttt{is}\ T \text{<} \overline{\lambda} \text{>} \rightsquigarrow \alpha', K', \{\alpha'\} \cup V_1}$$

Table 9. Command

$$\frac{\begin{array}{c}\Delta, x:(U,\lambda_1) \vdash e : U' \rightsquigarrow \lambda_2, K_1, V_1 \\ K' = K_1 \cup \{\lambda_2 \leq \lambda_1, \alpha_3 \leq \lambda_1\} \cup tcomp(U', U) \\ \alpha_3, \alpha_4 \notin V_1\end{array}}{P, \Delta, x:(U,\lambda_1), V \vdash x := e; \rightsquigarrow \texttt{com}\,(\alpha_3, \alpha_4), K', \{\alpha_3, \alpha_4\} \cup V_1}$$

$$\begin{array}{c}P, \Delta, x:(U,\lambda), V \vdash x := \texttt{new}\, U'(); \rightsquigarrow \texttt{com}\,(\alpha_1, \alpha_2), K, V' \\ K = \{\alpha_1 \leq \lambda\} \cup tcomp(U', U) \qquad V' = \{\alpha_1, \alpha_2\} \cup V \\ \alpha_1, \alpha_2 \notin V\end{array}$$

Method invocation and field update please see table 4 in the paper.

$$\frac{\begin{array}{c}\Delta, V \vdash e : \texttt{bool} \rightsquigarrow \lambda, K_1, V_1 \\ P, \Delta, V_1 \vdash S_1 \rightsquigarrow \texttt{com}\,(\lambda_1, \lambda_2), K_2, V_2 \\ P, \Delta, V_2 \vdash S_2 \rightsquigarrow \texttt{com}\,(\lambda_3, \lambda_4), K_3, V_3 \\ K = K_1 \cup K_2 \cup K_3 \cup \{\lambda \leq \lambda_1, \lambda \leq \lambda_2, \lambda \leq \lambda_3, \lambda \leq \lambda_4\} \\ \cup\{\alpha_5 \leq \lambda_1, \alpha_5 \leq \lambda_3, \alpha_6 \leq \lambda_2, \alpha_6 \leq \lambda_4\} \\ V' = \{\alpha_5, \alpha_6\} \cup V_3 \qquad \alpha_5, \alpha_6 \notin V_3\end{array}}{P, \Delta, V \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}\ \rightsquigarrow \texttt{com}\,(\alpha_5, \alpha_6), K', V'}$$

$$\frac{\begin{array}{c}\Delta, V \vdash e : U' \rightsquigarrow \lambda_1, K_1, V_1 \\ P, \Delta, x:(U,\lambda_2), V_1 \vdash S \rightsquigarrow \texttt{com}\,(\lambda_3, \lambda_4), K_2, V_2 \\ K' = K_1 \cup K_2 \cup tcomp(U', U) \cup \{\lambda_1 \leq \alpha_2; \alpha_5 \leq \lambda_3, \alpha_6 \leq \lambda_4\} \\ \alpha_2, \alpha_5, \alpha_6 \notin V_2\end{array}}{P, \Delta, V \vdash U\ x := e\ \texttt{in}\ S \rightsquigarrow \texttt{com}\,(\alpha_5, \alpha_6), K', \{\alpha_2, \alpha_5, \alpha_6\} \cup V_2}$$

$$\frac{\begin{array}{c}P - P' \cap Auth(\Delta^\dagger \texttt{self}), \Delta, V \vdash S \rightsquigarrow \texttt{com}\,(\lambda_1, \lambda_2), K_1, V_1 \\ K' = K_1 \cup \{\alpha_3 \leq \lambda_1, \alpha_4 \leq \lambda_2\} \qquad \alpha_3, \alpha_4 \notin V_1\end{array}}{P \vdash \texttt{enable}\ P'\ \texttt{in}\ S \rightsquigarrow \texttt{com}\,(\alpha_3, \alpha_4), K', \{\alpha_3, \alpha_4\} \cup V_1}$$

$$\frac{\begin{array}{c}P' \cap P \neq \emptyset \vee P' \nsubseteq Auth(\Delta^\dagger \texttt{self}) \\ P, \Delta, V \vdash S_2 \rightsquigarrow \texttt{com}\,(\lambda_3, \lambda_4), K_2, V_2\end{array}}{\begin{array}{c}P, \Delta, V \vdash \texttt{test}\ P'\texttt{then}\ S_1\texttt{else}\ S_2 \rightsquigarrow \texttt{com}\,(\alpha_5, \alpha_6), K', V' \\ K' = K_1 \cup \{\alpha_5 \leq \lambda_3, \alpha_6 \leq \lambda_4\} \\ V' = \{\alpha_5, \alpha_6\} \cup V_2 \qquad \alpha_5, \alpha_6 \notin V_2\end{array}}$$

$$\frac{\begin{array}{c}P' \cap P = \emptyset \qquad P' \subseteq Auth(\Delta^\dagger \texttt{self}) \\ P, \Delta, V \vdash S_2 \rightsquigarrow \texttt{com}\,(\lambda_1, \lambda_2), K_1, V_1 \\ P, \Delta, V_1 \vdash S_2 \rightsquigarrow \texttt{com}\,(\lambda_3, \lambda_4), K_2, V_2 \\ K' = K_1 \cup K_2 \cup \{\alpha_5 \leq \lambda_1, \alpha_5 \leq \lambda_3, \alpha_6 \leq \lambda_2, \alpha_6 \leq \lambda_4\} \\ V' = \{\alpha_5, \alpha_6\} \cup V_2 \qquad \alpha_5, \alpha_6 \notin V_2\end{array}}{P, \Delta, V \vdash \texttt{test}\ P'\texttt{then}\ S_1\texttt{else}\ S_2 \rightsquigarrow \texttt{com}\,(\alpha_5, \alpha_6), K', V'}$$