A Simple Semantics and Static Analysis for Stack Inspection

Anindya Banerjee
IMDEA Software Institute*
Madrid, Spain
anindya.banerjee@imdea.org

David A. Naumann
Stevens Institute of Technology†
Hoboken, NJ 07030, USA
naumann@cs.stevens.edu

The Java virtual machine and the .NET common language runtime feature an access control mechanism specified operationally in terms of run-time stack inspection. We give a denotational semantics in "eager" form, and show that it is equivalent to the "lazy" semantics using stack inspection. We give a static analysis of safety, *i.e.*, the absence of security errors, that is simpler than previous proposals. We identify several program transformations that can be used to remove run-time checks. We give complete, detailed proofs for safety of the analysis and for the transformations, exploiting compositionality of the eager semantics.

1 Introduction

System security depends in part on protecting resources through specified access control policies. For example, a policy may allow only some users the privilege to write the password file. A typical implementation of the policy found *e.g.*, in UNIX operating systems, involves an access control list $\mathscr{A}$ which associates with each user name n their set of privileges $\mathscr{A}(n)$. When a program is running it has an associated user, normally the user who invoked the program. To write a file, a program for user n must make a system call, and that system code checks whether $\mathscr{A}(n)$ includes the privilege of writing to the file. In order for users to be able to change their passwords, the system code for this task executes in a special mode ("setuid" in UNIX); the effective user is the owner of the code (say, *root*) rather than the originator of the call (n, which can write some files but not the password file).

The Java and .NET platforms offer a similar but more general security system [12, 13]. Instead of code being owned by a user or by "the system", there can be code from a number of sources, called *principals*, which can be offered varying degrees of trust. Moreover, instead of associating a principal with a loadable executable file, principals can be associated with fragments such as class declarations. Another refinement is that privileges must be explicitly enabled, by an operation called `doPrivileged`. The intent is that a program only enables its privileges when they are needed; this "principle of least privilege" [12] may help isolate the effect of security bugs and may facilitate static analysis. Before executing a dangerous operation, a check is made that the associated privilege has been enabled and is authorized for the current principal. This check is specified in terms of an implementation called *stack inspection*. Each stack frame is marked with the principal associated with the code for that frame, and the frame also records the privileges that have been enabled. This is used by procedure `checkPermission` which inspects the current stack.

The above description of the security system is an operational one. While the mechanism itself is easy to understand, it may over-constrain implementations, and it is difficult to analyze. Analysis is of interest, e.g., to determine whether a program can exhibit security exceptions when given its expected

*Partially supported by NSF grant EIA-980635, by Madrid Regional Government MINECO Project TIN2009-14599-C03-02 Desafios and EU NoE Project 256980 Nessos.
†Partially supported by NSF grants INT-9813854 and CNS-1228930.

A. Banerjee, O. Danvy, K.-G. Doh, J. Hatcliff (Eds):
David A. Schmidt's 60th Birthday Festschrift
EPTCS 129, 2013, pp. 284–308, doi:10.4204/EPTCS.129.17

permissions. Implementations of procedure calls do not always push stack, e.g., owing to inlining or tail call optimization. To understand the security properties achieved, and to optimize performance, we need analyses that capture the security model more abstractly.

Our contribution is threefold: (i) We give a denotational semantics in "eager" form, and show that it is equivalent to the "lazy" semantics using stack inspection. (ii) We give a static analysis of *safety, i.e.*, the absence of security errors. (iii) We identify several program transformations including some that can be used to remove run-time checks. We give detailed proofs for the analysis and for the transformations, exploiting compositionality of the eager semantics and simplicity of program equivalence in denotational semantics.

Related work. Skalka and Smith [17] give an operational semantics and use it to justify a static analysis of safety specified by a type system. Their type system is complicated by the choice of using a constraint system which is the basis for a type inference algorithm. We use a similar type system, but prefer to separate the specification of an analysis from algorithms to perform the analysis. We also include recursion in the language. Their semantics is easily seen to model the operational descriptions of stack inspection, but it has the usual shortcomings of operational semantics; for example, proofs ultimately go by induction on computations.

Wallach, Appel and Felten [19] model the mechanism with an operational semantics that manipulates formulas in a logic of authentication [1]. They show that the particular logical deductions corresponding to `checkPermission` can be decided efficiently, and propose an implementation called "security passing style" in which the security state is calculated in advance. The only result proven is equivalence of the two implementations. They do not include recursion or higher order functions, and the formal semantics is not made explicit. Although the use of logic sheds some light on the security properties achieved by the mechanism, the approach requires a considerable amount of theory that is not directly germane to analyzing safety or justifying optimizations.

Security passing style can be seen as a presentation of the eager means of evaluating security checks mentioned by Li Gong [12]. The eager semantics facilitates proofs, but JVM and CLR implementations use lazy semantics which appears to have better performance [12, 19, 13].

Pottier *et al.* [15] formalize the eager semantics by a translation into a lambda calculus, λ_{sec}, with operations on sets. Using an operational semantics for the calculus, a proof is sketched of equivalence with stack semantics. Using a very general framework for typing, a static analysis is given and a safety result is sketched. The language extends the language of Skalka and Smith [17] by adding permission tests. The works [17] and [15] aim to replace dynamic checks by static ones, but do not consider program transformations as such.

This paper originated as a technical report more than a decade ago [3]. At about the same time, and independently, Fournet and Gordon [11] investigated an untyped variant of λ_{sec}. They develop an equational theory that can be used to prove the correctness of code optimizations in the presence of stack inspection. A prime motivation for their work was the folklore that well-known program optimizations such as inlining and tail call elimination are invalidated by stack inpection. Their technical development uses small-step operational semantics and contextual equivalence of programs. To prove an extensive collection of program equivalences they develop a form of applicative bisimilarity. Additionally, they prove the equivalence of the lazy and eager implementations of stack inspection. This equivalence is also proved in Skalka's Ph.D. dissertation [16, Theorem 4.1, Chapter 4], where lazy and eager are termed "backward" and "forward" stack inspection respectively.

Where Fournet and Gordon point out how tail call elimination can be invalidated by stack inspection,

Clements and Felleisen [10] consider program semantics at the level of an abstract machine, namely the CESK machine. With this semantics they are able to show tail call optimization can be validated in full generality, and with its expected space savings. This is explored further by Ager et al [2]. They inter-derive a reduction semantics for the untyped variant of the λ_{sec}-calculus, an abstract machine, and a natural semantics, both without and with tail-call optimization. By unzipping the context in the abstract machine, they connect these semantics to Wallach *et al.*'s security passing style, characterize stack inspection as a computational monad, and combine this monad with the monad of exceptions.[1]

We treat a simply typed language similar to λ_{sec}, but with recursion. In contrast to the cited works we use a denotational semantics, which is straightforward; in fact, once the meanings of types are specified, the rest of the specification (*i.e.*, meanings of expressions) follows easily.[2] The simplicity of our model makes it possible to give a self-contained formal semantics and succinct but complete formal proofs. For example, denotational equality is a congruence simply because the semantics is compositional. We have not formally connected the semantics with an operational one. Adequacy seems obvious. Full abstraction is not obvious, but we have proved many of the contextual equivalences of Fournet and Gordon [11] and expect the remainder to be straightforward to show.

In addition to considering program transformations, Fournet and Gordon [11] address the question of what security properties can be enforced by stack inspection. They consider a variation that tracks history in the sense of what code has influenced the result of a computation. Pistoia et al [14] propose a variation that tracks implicit influences as well. The authors [5] propose another combination of information flow tracking with stack inspection, using a type and effect system where security types for functions are dependent on available permissions. In the interim, other code-based access mechanisms have been introduced (e.g., static permissions in the Android platform) and there have been further development of static analyses for security properties based on linear temporal logic [7, 6, 8, 18], but there seems to be little additional work on program equivalence in the presence of stack inspection.

Outline. The next section explains stack inspection informally, and it introduces our language. Section 3 gives the eager denotational semantics. Section 4 gives the static analysis for safety, including examples and correctness proof. Section 5 proves a number of representative program transformations. Section 6 shows how all checks can be removed from safe programs. Section 7 gives the stack-based denotational semantics and shows that it is equivalent to the eager semantics. Section 8 concludes.

2 Overview and language

Each declared procedure is associated with a principal n. We call n the *signer*, and write `signs` n e for a signed expression, because typically n is given by a cryptographic signature on a downloaded class file. During execution, each stack frame is labeled with the principal that signs the function, as well as the set P of privileges that have been explicitly enabled during execution of the function. For our purposes, a frame is a pair $\langle n, P\rangle$, and a nonempty stack is a list $\langle n, P\rangle :: S$ with $\langle n, P\rangle$ the top. There should be an initial stack $S_0 = \langle n_0, \varnothing\rangle :: nil$ for some designated n_0. An expression is evaluated in a stack S and with an environment h that provides values for its free variables.

Java provides operations to enable and disable a privilege, *i.e.*, to add it to the stack frame or remove it. Normally these are used in bracketed fashion, as provided by procedure `doPrivileged` which is given a privilege p and an expression e to evaluate. It enables p, evaluates e, and then disables p. Our

[1] Thanks to Olivier Danvy for communicating this explanation.

[2] Adding state is straightforward [4], but here we follow the cited works and confine attention to applicative expressions.

construct is written `dopriv` p `in` e. The effect of `dopriv` p `in` e in stack $\langle n, P\rangle :: S$ is to evaluate e in stack $\langle n, P \cup \{p\}\rangle :: S$, that is, to assert p in the current frame. This is done regardless of whether p is authorized for n.

Java's `checkPermission` operation checks whether a certain privilege has been enabled and is authorized for the current principal. Checking is done by inspecting the current stack. Each dangerous code fragment should be guarded by a check for an associated privilege, so that the code cannot be executed unless the check has succeeded. (This can be assured by inspection of the code, or by other forms of analysis [9] but is beyond the scope of our paper.) In our syntax, a guarded expression is written `check` p `for` e. The execution of an expression checked for privilege p is to raise a security error, which we denote by $\star$, unless the following predicate is true of p and the current stack.

$$\begin{array}{rcl} \mathsf{chk}(p, nil) & \Leftrightarrow & \mathsf{false} \\ \mathsf{chk}(p, (\langle n, P\rangle :: S)) & \Leftrightarrow & p \in \mathscr{A}(n) \wedge (p \in P \vee \mathsf{chk}(p, S)) \end{array}$$

That is, a privilege is enabled for a stack provided there is some frame $\langle n, P\rangle$ with $p \in P$ and p is authorized for n and is authorized for all principals in frames below this one.

A direct implementation in these terms requires inspecting some or all of the stack frames. The implementation is lazy in that no checking is performed when a privilege is enabled, only when it is needed to actually perform a guarded operation. On the other hand, each check incurs a significant cost, and in secure code the checks will never fail. Static analysis can detect unnecessary checks, and justify security-preserving transformations.

A stack S determines a set $\mathsf{privs}\ S$ of enabled, authorized privileges, to wit:

$$p \in \mathsf{privs}\ S \Leftrightarrow \mathsf{chk}(p, S)$$

This gives rise to a simple form of eager semantics: instead of evaluating an expression in the context of a stack S, we use $\mathsf{privs}\ S$, along with the current principal which appears on top of S. The eager, stack-free semantics is given in Section 3 and we will use this semantics exclusively in the static analysis and program transformations that follow in Sections 4 and 5.

A denotational semantics that uses explicit stacks will be deferred until Fig. 4 of Section 7. As mentioned previously, we will then take up the equivalence of the two semantics.

The language constructs are strict in $\star$: if a subexpression raises a security error, so does the entire expression. In Java, security errors are exceptions that can be caught. Thus it is possible for a program to determine whether a `checkPermission` operation will succeed. Rather than model the full exception mechanism, we include a construct `test` p `then` e_1 `else` e_2 which evaluates e_1 if $\mathsf{chk}(p, S)$ succeeds in the current stack S, and evaluates e_2 otherwise. Note that security error $\star$ is raised only by the `check` construct, not by `test` or `dopriv`.

In Java, the call of a procedure of a class signed by, or otherwise associated with, n, results in a new stack frame for the method, marked as owned by n. We model methods as function abstractions, but whereas Skalka and Smith use signed abstractions, we include a separate construct `signs` n e.[3] Evaluation of `signs` n e in stack S goes by evaluating e in stack $(\langle n, \varnothing\rangle :: S)$. For example, given a stack S, the evaluation of the application

$$(\texttt{fun}\ x.\ \texttt{signs}\ \mathit{user}\ \mathit{writepass}(x))\text{“myName”}$$

[3] Fournet and Gordon [11] also use a freely applicable construct for signing. Moreover they identify principals with sets of permissions: their "framed" expression $R[e]$ is like our `signs` n e for n with $\mathscr{A}(n) = R$.

amounts to evaluating $writepass$("myName") in the stack $(\langle user, \varnothing\rangle :: S)$.

We separate `signs` from abstractions because it helps disentangle definitions and proofs, *e.g.*, these constructs are treated independently in our safety result. On the other hand, unsigned abstractions do not model the Java mechanism. In our consistency result, Theorem 7.2, we show that our semantics is equivalent to stack inspection for all *standard expressions*, *i.e.*, those in which the body of every abstraction is signed.

2.1 Syntax and typing

Given are sets Principals and Privileges, and a fixed access control list $\mathscr{A}$ that maps Principals to sets of privileges. In the grammar for data types and expressions, n ranges over Principals and p over Privileges. Application associates to the left. We include recursive definitions for expressiveness, and simple abstractions $\texttt{fun}\, x.\ e$ which, while expressible using `letrec`, are easier to understand in definitions and proofs. For simplicity, the only primitive type is `bool`, and the only type constructor is for functions. Products, sums, and other primitive types can be added without difficulty. Throughout the paper we use `true` to exemplify the treatment of constants in general.

$$
\begin{array}{rcl}
t & ::= & \texttt{bool} \mid (t \to t) \\
e & ::= & \texttt{true} \mid x \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\
 & & \texttt{fun}\ x.\ e \mid e_1\ e_2 \mid \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e_2 \mid \\
 & & \texttt{signs}\ n\ e \mid \texttt{dopriv}\ p\ \texttt{in}\ e \mid \texttt{check}\ p\ \texttt{for}\ e \mid \texttt{test}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2
\end{array}
$$

A signed abstraction ${}^n\lambda x.e$ in the language of Skalka and Smith is written $\texttt{fun}\, x.\ \texttt{signs}\ n\ e$ in ours. Our safety result can be proved without restriction to expressions of this form. But for the eager semantics to be equivalent to stack semantics, it is crucial that function bodies be signed so the semantics correctly tracks principals on behalf of which the body of an abstraction is evaluated.

Definition 2.1 (Standard expression) An expression is standard if for every subexpression $\texttt{fun}\, x.\ e$ or $\texttt{letrec}\, f(x) = e\ \texttt{in}\ e_1$ we have that e is $\texttt{signs}\ n\ e'$ for some n, e'.

Well-formed expressions are characterized by typing judgements $D \vdash e : t$ which express that e has type t where free identifiers are declared by D. A typing context D is a labeled tuple of declarations $\{x_1 : t_1, \ldots, x_k : t_k\}$. We write $D, x : t$ for the extended context $\{x_1 : t_1, \ldots, x_k : t_k, x : t\}$, and $D.x_i$ for the type of x_i. The typing rules are given in Figure 1.

2.2 The password example

As an example of the intended usage, we consider the problem of protecting the password file, using a privilege p for changing password and w for writing to the password file. The user is authorized to change passwords: $\mathscr{A}(user) = \{p\}$. Root is authorized to change passwords and to write the password file: $\mathscr{A}(root) = \{p, w\}$. Suppose $hwWrite$ is the operating system call which needs to be protected from direct user access. The system provides the following code, which guards $hwWrite$ with the privilege w.

$$
\begin{array}{rcl}
writepass & = & \texttt{fun}\, x.\ \texttt{signs}\ root\ \texttt{check}\ w\ \texttt{for}\ hwWrite(x, \text{“/etc/password”}) \\
passwd & = & \texttt{fun}\, x.\ \texttt{signs}\ root\ \texttt{check}\ p\ \texttt{for}\ \texttt{dopriv}\ w\ \texttt{in}\ writepass(x)
\end{array}
$$

$$D \vdash \texttt{true} : \texttt{bool} \qquad D, x : t \vdash x : t$$

$$\frac{D \vdash e : \texttt{bool} \qquad D \vdash e_1 : t \qquad D \vdash e_2 : t}{D \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t}$$

$$\frac{D, x : t_1 \vdash e : t_2}{D \vdash \texttt{fun}\ x.\ e : t_1 \to t_2} \qquad \frac{D \vdash e_1 : t_1 \to t_2 \qquad D \vdash e_2 : t_1}{D \vdash e_1\ e_2 : t_2}$$

$$\frac{D, f : t_1 \to t_2, x : t_1 \vdash e_1 : t_2 \qquad D, f : t_1 \to t_2 \vdash e_2 : t}{D \vdash \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e_2 : t}$$

$$\frac{D \vdash e : t}{D \vdash \texttt{signs}\ n\ e : t} \qquad \frac{D \vdash e : t}{D \vdash \texttt{dopriv}\ p\ \texttt{in}\ e : t}$$

$$\frac{D \vdash e : t}{D \vdash \texttt{check}\ p\ \texttt{for}\ e : t} \qquad \frac{D \vdash e_1 : t \qquad D \vdash e_2 : t}{D \vdash \texttt{test}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t}$$

Figure 1: Typing rules.

Consider the following user programs.

$$\begin{array}{lcl} bad1 & = & \texttt{signs}\ user\ writepass(\text{“mypass”}) \\ bad2 & = & \texttt{signs}\ user\ \texttt{dopriv}\ w\ \texttt{in}\ writepass(\text{“mypass”}) \\ use & = & \texttt{signs}\ user\ \texttt{dopriv}\ p\ \texttt{in}\ passwd(\text{“mypass”}) \end{array}$$

Here $bad1$ raises a security exception because $writepass$ checks for privilege w which is not possessed by $user$. The user can try to enable w, as in $bad2$, but because w is not authorized for $user$ the exception is still raised. By contrast, use does not raise an exception: function $passwd$ checks for privilege p which is possessed by $user$, and it enables the privilege w needed by $writepass$. Using transformations discussed in Section 5, checks that never fail can be eliminated. For example, the analysis will show that use is safe, and the transformations will reduce use to

$$\texttt{signs}\ user\ \texttt{signs}\ root\ hwWrite(\text{“mypass”}, \text{“/etc/password”})$$

3 Denotational semantics

This section gives the eager denotational semantics.

3.1 Meanings of types and type contexts

A *cpo* is a partially ordered set with least upper bounds of ascending chains; it need not have a least element. Below we define, for each type t, a cpo $[\![t]\!]$. We assume that $\bot$ and $\star$ are two values not in $\{\mathsf{true}, \mathsf{false}\}$ and not functions; this will ensure that $\{\bot, \star\} \cap [\![t]\!] = \varnothing$ for all t. We will identify $\bot$ with non-termination and $\star$ with security errors. For cpo C, define $C_{\bot\star} = C \cup \{\bot, \star\}$, ordered as the disjoint union of C with $\{\star\}$, lifted with $\bot$. That is, for any $u, v \in C_{\bot\star}$, define $u \leq v$ iff $u = \bot$, $u = v$, or u and v are in C and $u \leq v$ in C.

We define $[\![\texttt{bool}]\!] = \{\mathsf{true}, \mathsf{false}\}$, ordered by equality. We also take the powerset $\mathscr{P}(\mathsf{Privileges})$ to be a cpo ordered by equality. Define

$$[\![t_1 \to t_2]\!] \;=\; \mathscr{P}(\mathsf{Privileges}) \to [\![t_1]\!] \to [\![t_2]\!]_{\bot\star}$$

where $\rightarrow$ associates to the right and denotes continuous function space, ordered pointwise. Note that lubs are given pointwise. Also, $[\![t_1 \rightarrow t_2]\!]$ does not contain $\perp$ but it does have a least element, namely the constant function $\lambda\, P.\, \lambda\, d.\, \perp$.

Principals behave in a lexically scoped way. By contrast, privileges are dynamic and vary during execution; this is reflected in the semantics of the function type.

Let $D = \{x_1 : t_1, \ldots, x_k : t_k\}$ be a type context. Then $[\![D]\!]$ is defined to be the set $\{x_1 : [\![t_1]\!], \ldots, x_k : [\![t_k]\!]\}$ of labeled tuples of appropriate type. If h is such a record, we write $h.x_i$ for the value of field x_i. If D is the empty type context $\varnothing$, then the only element of $[\![D]\!]$ is the empty record $\{\}$. For $h \in [\![D]\!]$ and $d \in [\![t]\!]$ we write $[h \mid x \mapsto d]$ for the extended record in $[\![D, x : t]\!]$.

3.2 Meanings of expressions

An expression judgement denotes a function

$$[\![D \vdash e : t]\!] \in \mathsf{Principals} \rightarrow \mathscr{P}(\mathsf{Privileges}) \rightarrow [\![D]\!] \rightarrow [\![t]\!]_{\perp\star}$$

Given a principal n, a set $P \in \mathscr{P}(\mathsf{Privileges})$ denoting privileges required by e, and environment $h \in [\![D]\!]$, the meaning of $[\![D \vdash e : t]\!] n P h$ is either $\perp$ or $\star$ or an element of $[\![t]\!]$.

In contrast with the work of Fournet and Gordon we do not restrict P to be a subset of $\mathscr{A}(n)$, though it can easily be done —simply by giving the denotation this dependent type:

$$[\![D \vdash e : t]\!] \in (n : \mathsf{Principals}) \rightarrow \mathscr{P}(\mathscr{A}(n)) \rightarrow [\![D]\!] \rightarrow [\![t]\!]_{\perp\star} \tag{1}$$

In programs of interest, signed at the top level, most expressions will in fact be applied to permission sets that satisfy the restriction. Later we observe that the restriction is need for validity of some transformations, but surprisingly few of them.

In the denotational semantics (Figure 2), we use the metalanguage construct, **let** $d = E_1$ **in** E_2, with the following semantics: if the value of E_1 is either $\perp$ or $\star$ then that is the value of the entire let expression; otherwise, its value is the value of E_2 with d bound to the value of E_1. The semantics of if-then-else is $\star$-strict in the guard. We also write $P \sqcup_n \{p\}$ for **if** $p \in \mathscr{A}(n)$ **then** $P \cup \{p\}$ **else** P.

The semantics is standard for the most part. We will only explain the meanings of the expressions that directly concern security. In what follows, we will assume, unless otherwise stated, that expression e is signed by principal n and is computed with privilege set P and in environment h.

The meaning of `signs` n' e is the meaning of e, signed by n', computed with privilege set $P \cap \mathscr{A}(n')$, in h. To illustrate the idea, consider Li Gong's example [12, Section 3.11.2]. A game applet, *applet*, has a method that calls *FileInputStream* to open the file containing the ten current high scores. In our semantics, this scenario entails finding the meaning of `signs` $system\ FileInputStream$, as invoked under some privilege set $P \subseteq \mathscr{A}(applet)$; and, this means we need to find the meaning of *FileInputStream* (*i.e.*, whether read privileges are enabled) under the privilege set $P \cap \mathscr{A}(system)$. Assuming *system* has all privileges, this reduces to checking if *applet* has been granted permission to read. If it has not been granted the permission, the file will not be read, even though it calls system code to do so.

The meaning of `dopriv` p `in` e is the meaning of e computed with privilege set $P \cup \{p\}$ if $p \in \mathscr{A}(n)$, and is the meaning of e computed with privilege set P if $p \notin \mathscr{A}(n)$. The meaning of `check` p `for` e is a security error if $p \notin P$; otherwise, the meaning is that of e. Finally, the meaning of `test` p `then` e_1 `else` e_2 is the meaning of e_1 or e_2 according as $p \in P$ or $p \notin P$.

We leave it to the reader to check that the semantics of each construct is a continuous function of the semantics of its constituent expressions, so the semantics of recursion is well defined.

$$
\begin{array}{lcl}
[\![D \vdash \texttt{true} : \texttt{bool}]\!] nPh & = & \mathsf{true} \\
[\![D, x : t \vdash x : t]\!] nPh & = & h.x \\
[\![D \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t]\!] nPh & = & \textbf{let}\ b\ =\ [\![D \vdash e : \texttt{bool}]\!] nPh\ \textbf{in} \\
 & & \textbf{if}\ b\ \textbf{then}\ [\![D \vdash e_1 : t]\!] nPh\ \textbf{else}\ [\![D \vdash e_2 : t]\!] nPh \\
[\![D \vdash \texttt{fun}\ x.\ e : t_1 \to t_2]\!] nPh & = & \lambda P' \in \mathscr{P}(\mathsf{Privileges}).\ \lambda d \in [\![t_1]\!]. \\
 & & [\![D, x : t_1 \vdash e : t_2]\!] nP'[h \mid x \mapsto d] \\
[\![D \vdash e_1\ e_2 : t_2]\!] nPh & = & \textbf{let}\ f\ =\ [\![D \vdash e_1 : t_1 \to t_2]\!] nPh\ \textbf{in} \\
 & & \textbf{let}\ d = [\![D \vdash e_2 : t_1]\!] nPh\ \textbf{in}\ fPd \\
[\![D \vdash \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e_2 : t]\!] nPh & & \\
\quad = \ \textbf{let}\ G(g)\ =\ \lambda P'.\ \lambda d.\ [\![D, f : t_1 \to t_2, x : t_1 \vdash e_1 : t_2]\!] nP'[h \mid f \mapsto g, x \mapsto d]\ \textbf{in} & & \\
\quad [\![D, f : t_1 \to t_2 \vdash e_2 : t]\!] nP[h \mid f \mapsto \mathit{fix}\ G] & & \\
[\![D \vdash \texttt{signs}\ n'\ e : t]\!] nPh & = & [\![D \vdash e : t]\!] n'(P \cap \mathscr{A}(n'))h \\
[\![D \vdash \texttt{dopriv}\ p\ \texttt{in}\ e : t]\!] nPh & = & [\![D \vdash e : t]\!] n(P \sqcup_n \{p\})h \\
[\![D \vdash \texttt{check}\ p\ \texttt{for}\ e : t]\!] nPh & = & \textbf{if}\ p \in P\ \textbf{then}\ [\![D \vdash e : t]\!] nPh\ \textbf{else}\ \star \\
[\![D \vdash \texttt{test}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t]\!] nPh & = & \textbf{if}\ p \in P\ \textbf{then}\ [\![D \vdash e_1 : t]\!] nPh\ \textbf{else}\ [\![D \vdash e_2 : t]\!] nPh
\end{array}
$$

Figure 2: Denotational semantics

4 Static Analysis

The denotational semantics in Section 3 gives a dynamic or run-time view of safety; if a program is safe, its execution will not yield $\star$. In this section, we specify a type system that statically guarantees safety; if a program is well-typed in the system then it is safe. One may utilize the static analysis for optimizing programs *e.g.*, removing redundant checks of privileges at run-time.

The static analysis is specified by an extended form of typing judgement. The idea is to give not only the type of an expression, but a principal n and set P of privileges for which the expression is safe. An arrow type $t_1 \to t_2$ denotes functions dependent on a set of privileges, and the static analysis uses annotated types to track sets of privileges adequate for safety. We adopt a Greek notational style for types in the static analysis. Letting Π range over sets of privileges, annotated types, θ, are defined by

$$\theta ::= \texttt{bool} \mid (\theta_1 \xrightarrow{\Pi} \theta_2)$$

For this syntax to be finitary, one could restrict Π to finite sets, but we have no need for such restriction in our proofs. An expression typed $\theta_1 \xrightarrow{\Pi} \theta_2$ signifies that its application may require at least the privileges Π for safe execution.

4.1 Type-based analysis

The analysis is specified by the typing judgement $\Delta;\ n \vdash e : \theta,\ \Pi$. In words, expression e signed by principal n and typed in context Δ, has (annotated) type θ and is safe provided at least the set Π of privileges are enabled. Figure 3 gives the specification.

Constant `true`, identifiers, and anonymous functions of the form $\texttt{fun}\ x.\ e$ are all safe: they do not require any privileges be enabled for safe execution. However, the body e in $\texttt{fun}\ x.\ e$, may require a set

$$\Delta;\ n \vdash \texttt{true} : \texttt{bool},\ \varnothing$$

$$\Delta, x : \theta;\ n \vdash x : \theta,\ \varnothing$$

$$\frac{\Delta, x : \theta_1;\ n \vdash e : \theta_2,\ \Pi}{\Delta;\ n \vdash \texttt{fun}\ x.\ e : \theta_1 \xrightarrow{\Pi} \theta_2,\ \varnothing}$$

$$\frac{\Delta;\ n \vdash e_1 : \theta_1 \xrightarrow{\Pi} \theta_2,\ \Pi_1 \qquad \Delta;\ n \vdash e_2 : \theta_1',\ \Pi_2 \qquad \theta_1' \leq \theta_1}{\Delta;\ n \vdash e_1\ e_2 : \theta_2,\ \Pi \cup \Pi_1 \cup \Pi_2}$$

$$\frac{\Delta;\ n \vdash e : \texttt{bool},\ \Pi_1 \qquad \Delta;\ n \vdash e_1 : \theta,\ \Pi_2 \qquad \Delta;\ n \vdash e_2 : \theta,\ \Pi_3}{\Delta;\ n \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \theta,\ \Pi_1 \cup \Pi_2 \cup \Pi_3}$$

$$\frac{\Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2, x : \theta_1;\ n \vdash e_1 : \theta_2,\ \Pi \qquad \Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2;\ n \vdash e_2 : \theta,\ \Pi_1}{\Delta;\ n \vdash \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e_2 : \theta,\ \Pi \cup \Pi_1}$$

$$\frac{\Delta;\ n \vdash e : \theta,\ \Pi}{\Delta;\ n \vdash \texttt{check}\ p\ \texttt{for}\ e : \theta,\ \Pi \cup \{p\}} \qquad \frac{\Delta;\ n \vdash e : \theta,\ (\Pi \sqcup_n \{p\})}{\Delta;\ n \vdash \texttt{dopriv}\ p\ \texttt{in}\ e : \theta,\ \Pi}$$

$$\frac{\Delta;\ n' \vdash e : \theta,\ \Pi \qquad \Pi \subseteq \mathscr{A}(n')}{\Delta;\ n \vdash \texttt{signs}\ n'\ e : \theta,\ \Pi} \qquad \frac{\Delta;\ n \vdash e_1 : \theta,\ \Pi_1 \qquad \Delta;\ n \vdash e_2 : \theta,\ \Pi_2}{\Delta;\ n \vdash \texttt{test}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \theta,\ \Pi_1 \cup \Pi_2}$$

Figure 3: Static analysis

of privileges Π be enabled. This is manifest in the type $\theta_1 \xrightarrow{\Pi} \theta_2$. The latent privileges, Π, get exposed during an application, $e_1 e_2$. Say e_1 has type $\theta_1 \xrightarrow{\Pi} \theta_2$; if Π_1 may be enabled during e_1's execution, and Π_2 may be enabled during e_2's execution, then application itself may require Π be enabled; hence $\Pi \cup \Pi_1 \cup \Pi_2$ may be enabled during the execution of $e_1 e_2$. The application rule also uses subtyping, as discussed in the sequel.

The analysis for `check` p `for` e requires that in addition to privileges enabled for e, the privilege p be enabled so that the check is safe. If Π is the set of privileges that may be enabled during the execution of `dopriv` p `in` e, then p can be assumed to be enabled during the execution of e, provided $p \in \mathscr{A}(n)$.

Finally, for `signs` n' e the only privileges that should be enabled are the ones authorized for n'. Note that a signed expression can occur in a term with a different owner, so it is not the case that $\Pi \subseteq \mathscr{A}(n)$ for every derivable $\Delta; n \vdash e : \theta,\ \Pi$.

4.2 Subtyping

Where Skalka and Smith [17] use a constraint-based type system whose constraints subsequently must be solved,[4] our analysis is syntax-directed. In some sense, our system gives minimal types and privilege assumptions. (Pottier *et al.*'s system [15] enjoys a principal types property also. Reasoning about minimal security contexts for code invocation is also considered in several papers by Besson *et al.* [7, 6, 8].) We do not formalize this notion, but informally it sheds light on the specification of the analysis. In the case of values, such as variables and abstraction, the privilege set is empty. In the case of `check` p `for` e, the rule adds the checked privilege p to the "minimal" privileges of e, and similarly for the other security constructs. In the case of conditional, a union is formed from the "minimal" privileges of the constituent expressions, and the types of the constituents are the same as the type of the conditional. By contrast, in the case of application $e_1 e_2$, the "minimal" types and privileges for e_1 and e_2 need not match exactly. So we define a relation of subtyping with the informal meaning that $\theta' \leq \theta$ provided the privileges required

[4] Pottier *et al.* [15] use unification of row variables, in a relatively complicated system.

by θ' are contained in those required by θ. This is significant only in case e_2 has functional type, in which case the latent privileges of e_2 should be among those of e_1.

Subtyping is defined as the least relation $\leq$ with $\texttt{bool} \leq \texttt{bool}$ and, for arrow types, $\theta_1 \xrightarrow{\Pi_1} \theta_1' \leq \theta_2 \xrightarrow{\Pi_2} \theta_2'$ provided $\theta_2 \leq \theta_1$, $\theta_1' \leq \theta_2'$, and $\Pi_1 \subseteq \Pi_2$.

To relate the semantics to the static analysis, we need the ordinary type θ^* obtained by erasing annotations. This is defined by induction on θ, to wit: $\texttt{bool}^* = \texttt{bool}$ and $(\theta_1 \xrightarrow{\Pi} \theta_2)^* = \theta_1^* \to \theta_2^*$. It is easy to show that if $\theta_1 \leq \theta_2$, then $\theta_1^* = \theta_2^*$.

Due to subtyping, an expression can have more than one annotated type and satisfy more than one judgement. But a derivable judgement $\Delta;\ n \vdash e : \theta,\ \Pi$ has only one derivation, which is dictated by the structure of e. Proofs in the sequel will go by "induction on e", meaning induction on the derivation of some judgement $\Delta;\ n \vdash e : \theta,\ \Pi$.

4.3 Examples

For any n, the expressions in the password example (Section 2.2) can be analyzed as follows.

$$\begin{array}{l} \varnothing;\ n \vdash \mathit{writepass} : \mathit{string} \xrightarrow{\{w\}} \mathit{void},\ \varnothing \\ \varnothing;\ n \vdash \mathit{passwd} : \mathit{string} \xrightarrow{\{p\}} \mathit{void},\ \varnothing \\ \varnothing;\ n \vdash \mathit{use} : \mathit{void},\ \varnothing \end{array}$$

This confirms that use is safe. On the other hand, there is no Π such that $\varnothing;\ n \vdash \mathit{bad}1 : \mathit{void},\ \Pi$ or $\varnothing;\ n \vdash \mathit{bad}2 : \mathit{void},\ \Pi$. Such Π must satisfy $w \in \Pi$ for the application of $\mathit{writepass}$, owing to the rules for application and for `dopriv in` . And Π must satisfy $\mathscr{A}(\mathit{user}) \subseteq \Pi$ by the rule for signs.

Here is another example, inspired by ones in Skalka and Smith [17]. Define the following standard expressions:

$$\begin{array}{rcl} \mathit{lp} & = & \texttt{fun}\, f.\, \texttt{signs}\ n\ (\texttt{fun}\, x.\, \texttt{signs}\ n\ (\texttt{dopriv}\ p\ \texttt{in}\ (f\ x))) \\ \mathit{cp} & = & \texttt{fun}\, x.\, \texttt{signs}\ n\ (\texttt{check}\ p\ \texttt{for}\ x) \end{array}$$

The reader can verify that one analysis for cp is given by the typing $\Delta;\ n \vdash \mathit{cp} : (\texttt{bool} \xrightarrow{\{p\}} \texttt{bool}), \varnothing$ and that the typing demands $p \in \mathscr{A}(n)$. Similarly, the reader can verify that one possible analysis for lp is given by the typing $\Delta;\ n \vdash \mathit{lp} : (\texttt{bool} \xrightarrow{\{p\}} \texttt{bool}) \xrightarrow{\varnothing} (\texttt{bool} \xrightarrow{\varnothing} \texttt{bool}), \varnothing$.

For all $P \in \mathscr{P}(\mathsf{Privileges})$, for all $h : \Delta^*$, we can show (omitting types and some steps),

$$\begin{array}{rcl} [\![\mathit{lp}]\!] nPh & = & \lambda P_1.\, \lambda d_1.\, \lambda P_2.\, \lambda d_2.\, [\![\texttt{dopriv}\ p\ \texttt{in}\ f\ x]\!] n(P_2 \cap \mathscr{A}(n))[h \mid f \mapsto d_1, x \mapsto d_2] \\ & = & \{\text{letting } P_3 = P_2 \cap \mathscr{A}(n)\} \\ & & \lambda P_1.\, \lambda d_1.\, \lambda P_2.\, \lambda d_2.\, d_1(P_3 \sqcup_n \{p\})\, d_2 \\ [\![\mathit{cp}]\!] nPh & = & \lambda P_1'.\, \lambda d_1'.\, \textbf{if } p \in (P_1' \cap \mathscr{A}(n)) \textbf{ then } [\![x]\!] n(P_1' \cap \mathscr{A}(n))[h \mid x \mapsto d_1'] \textbf{ else } \star \\ & = & \lambda P_1'.\, \lambda d_1'.\, \textbf{if } p \in (P_1' \cap \mathscr{A}(n)) \textbf{ then } d_1' \textbf{ else } \star \end{array}$$

Let $F = [\![\mathit{lp}]\!] nPh$, let $d = [\![\mathit{cp}]\!] nPh$, and let $G = [\![(\mathit{lp}\ \mathit{cp})]\!] nPh$. Then

$$\begin{array}{rcl} [\![(\mathit{lp}\ \mathit{cp})]\!] nPh & = & FPd \\ & = & \lambda P_2.\, \lambda d_2.\, \textbf{if } p \in ((P_3 \sqcup_n \{p\}) \cap \mathscr{A}(n)) \textbf{ then } d_2 \textbf{ else } \star \\ & = & \lambda P_2.\, \lambda d_2.\, d_2 \quad \text{because } p \in \mathscr{A}(n) \\ [\![(\mathit{lp}\ \mathit{cp})\texttt{true}]\!] nPh & = & GP([\![\texttt{true}]\!] nPh) \\ & = & \mathsf{true} \end{array}$$

Hence $(\mathit{lp}\ \mathit{cp})\texttt{true}$ is safe in any environment and typable as $\Delta;\ n \vdash (\mathit{lp}\ \mathit{cp})\texttt{true} : \texttt{bool}, \varnothing$.

4.4 Safety of the analysis

Theorem 4.1 (Safety) Suppose $\varnothing;\ n \vdash e : \theta,\ \Pi$ is derivable. Then for all $P \in \mathscr{P}(\mathsf{Privileges})$ with $\Pi \subseteq P$, it is the case that $[\![\varnothing \vdash e : \theta^*]\!] n P\{\} \neq \star$.

Proof: Immediate consequence of Lemma 4.5 below. ■

In order to serve as an adequate induction hypothesis, the lemma strengthens the theorem by allowing judgements with non-empty contexts. But this is not enough. Values at arrow types are functions that depend on privilege sets. As induction hypothesis for the case of application we require these functions be safe with respect to the privilege set Π annotating their type.

Definition 4.2 For each annotated type θ the predicate $\mathsf{safe}\ \theta$ on $[\![\theta^*]\!]_{\perp\star}$ is defined as follows:
$\mathsf{safe}\ \theta(\perp) \Leftrightarrow \mathsf{true}$ and $\mathsf{safe}\ \theta(\star) \Leftrightarrow \mathsf{false}$ for all θ. For values other than $\perp$ and $\star$, the definition is by induction on structure of θ.

$$\begin{array}{lcl} \mathsf{safe}\ \mathtt{bool}(b) & \Leftrightarrow & \mathsf{true} \\ \mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(f) & \Leftrightarrow & \forall P \in \mathscr{P}(\mathsf{Privileges}).\forall d \in [\![\theta_1^*]\!]. \\ & & \Pi \subseteq P \wedge \mathsf{safe}\ \theta_1(d) \Rightarrow \mathsf{safe}\ \theta_2(fPd) \end{array}$$

The predicate $\mathsf{safe}\ \Delta$ on $[\![\Delta^*]\!]$ is defined by $\mathsf{safe}\ \Delta(h) \Leftrightarrow \forall x \in \mathit{dom}(h).\mathsf{safe}(\Delta.x)(h.x)$. Recall that $h.x \neq \perp$ and $h.x \neq \star$, because $\perp \notin [\![t]\!]$ and $\star \notin [\![t]\!]$, for all t.

Fact 4.3 $\theta \leq \theta'$ and $\mathsf{safe}\ \theta\ d$ imply $\mathsf{safe}\ \theta'\ d$.

Proof: By induction on derivation of $\theta \leq \theta'$. The result is clear for $\mathtt{bool} \leq \mathtt{bool}$. For $(\theta_1 \xrightarrow{\Pi} \theta_2) \leq (\theta_1' \xrightarrow{\Pi'} \theta_2')$, assume $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)\ f$. To show $\mathsf{safe}\ (\theta_1' \xrightarrow{\Pi'} \theta_2')\ f$, consider any $P \in \mathscr{P}(\mathsf{Privileges})$, such that $\Pi' \subseteq P$, and any $d \in [\![\theta_1'^*]\!]$ with $\mathsf{safe}\ \theta_1'\ d$. From the subtyping, we know that $\Pi \subseteq \Pi'$, hence $\Pi \subseteq P$. Moreover, by induction on derivation of $\theta_1' \leq \theta_1$, we obtain $\mathsf{safe}\ \theta_1'\ d$ implies $\mathsf{safe}\ \theta_1\ d$. Hence from assumption $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)\ f$, we obtain $\mathsf{safe}\ \theta_2(fPd)$ holds. Now by induction on derivation $\theta_2 \leq \theta_2'$, we obtain $\mathsf{safe}\ \theta_2'(fPd)$. ■

Lemma 4.4 The predicate safe preserves lubs. That is, for any θ, let $u : \mathbb{N} \to [\![\theta^*]\!]_{\perp\star}$ be an ascending chain. Then, $\forall i.\mathsf{safe}\ \theta\ (u_i)$ implies $\mathsf{safe}\ \theta\ (\bigsqcup_i u_i)$.

Proof: By structural induction on θ. When $\theta = \mathtt{bool}$, the assumption $\mathsf{safe}\ \theta\ (u_i)$ implies $u_i \neq \star$ for each i, so $\bigsqcup_i u_i$ is true or false or $\perp$. Thus the result holds by definition safe.
When $\theta = (\theta_1 \xrightarrow{\Pi} \theta_2)$, assume $P \in \mathscr{P}(\mathsf{Privileges})$ and $d \in [\![\theta_1^*]\!]$, such that $\Pi \subseteq P$ and $\mathsf{safe}\ \theta_1(d)$. Then, from assumption $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)\ u_i$ we obtain $\mathsf{safe}\ \theta_2\ (u_i P d)$ holds for every i. By the induction hypothesis on θ_2, we get $\mathsf{safe}\ \theta_2\ (\bigsqcup_i (u_i P d))$. Lubs are pointwise, so we get $\mathsf{safe}\ \theta_2\ ((\bigsqcup_i u_i) P d)$. ■

Lemma 4.5 Suppose $\Delta;\ n \vdash e : \theta,\ \Pi$ is derivable. Then for all $P \in \mathscr{P}(\mathsf{Privileges})$, for all $h \in [\![\Delta^*]\!]$, if $\mathsf{safe}\ \Delta(h)$ and $\Pi \subseteq P$ then $\mathsf{safe}\ \theta\ ([\![\Delta^* \vdash e : \theta^*]\!] n P h)$.

Theorem 4.1 follows from the lemma because $\mathsf{safe}\ \varnothing\{\}$ and $\mathsf{safe}\ \theta([\![\varnothing \vdash e : \theta^*]\!] n P\{\})$ implies $[\![\varnothing \vdash e : \theta^*]\!] n P\{\} \neq \star$.

Another consequence of the lemma is that the language admits additional constants at all types, declared in an initial context D_0, provided the corresponding initial environment assigns a safe meaning to each identifier in D_0.

Proof: of Lemma. Go by induction on the typing derivation, $\Delta;\ n \vdash e : \theta,\ \Pi$. Throughout, we assume $P \in \mathscr{P}(\mathsf{Privileges})$ and $h \in [\![\Delta^*]\!]$ and $\mathsf{safe}\ \Delta(h)$ and $\Pi \subseteq P$, and also let $u = [\![\Delta^* \vdash e : \theta^*]\!] nPh$ for each case of e.

- Case `true`: Then, $u = \mathsf{true}$ so $\mathsf{safe}\ \mathtt{bool}(u)$ by definition safe.
- Case x: Then, $u = h.x$ and $\mathsf{safe}\ \theta(h.x)$ follows, by definition safe, from the assumption $\mathsf{safe}\ \Delta(h)$.
- Case `if` e `then` e_1 `else` e_2: Then $\Pi_1 \cup \Pi_2 \cup \Pi_3 \subseteq P$, and

  $$u = \textbf{if}\ b\ \textbf{then}\ [\![\Delta^* \vdash e_1 : \theta^*]\!] nPh\ \textbf{else}\ [\![\Delta^* \vdash e_2 : \theta^*]\!] nPh$$

  where $b = [\![\Delta^* \vdash e : \mathtt{bool}]\!] nPh$. By the induction hypothesis on the typing derivation of e, noting that $\Pi_1 \subseteq P$, we have $\mathsf{safe}\ \mathtt{bool}(b)$ and hence $b \neq \star$. If $b = \bot$ then $u = \bot$ and $\bot$ is safe. Otherwise, $b = \mathsf{true}$ or $b = \mathsf{false}$. In the former case, by the induction hypothesis on the typing derivation of e_1, noting that $\Pi_2 \subseteq P$, we have $\mathsf{safe}\ \theta(u)$. The case of $b = \mathsf{false}$ is symmetric.
- Case `fun` $x.\ e$: Then $u = \lambda P'.\ \lambda d.\ [\![\Delta^*, x : \theta_1^* \vdash e : \theta_2^*]\!] nP'[h \mid x \mapsto d]$. Thus $u \neq \star$. To prove $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(u)$, consider any $P'' \in \mathscr{P}(\mathsf{Privileges})$ and any $d' \in [\![\theta_1^*]\!]$ such that $\Pi \subseteq P''$ and $\mathsf{safe}\ \theta_1(d')$, to show $\mathsf{safe}\ \theta_2(uP''d')$. By semantics, $uP''d' = [\![\Delta^*, x : \theta_1^* \vdash e : \theta_2^*]\!] nP''[h \mid x \mapsto d']$, so the induction hypothesis for e yields $\mathsf{safe}\ \theta_2(uP''d')$ provided that $\Pi \subseteq P''$ and $\mathsf{safe}\ (\Delta, x : \theta_1)[h \mid x \mapsto d']$. We have $\Pi \subseteq P''$ by assumption, and $\mathsf{safe}\ (\Delta, x : \theta_1)[h \mid x \mapsto d']$ follows from $\mathsf{safe}\ \Delta(h)$ and $\mathsf{safe}\ \theta_1(d')$.
- Case $e_1\ e_2$: Let $f = [\![\Delta^* \vdash e_1 : \theta_1^* \to \theta_2^*]\!] nPh$ and $d = [\![\Delta^* \vdash e_2 : \theta_1'^*]\!] nPh$, so that $u = fPd$. (Recall that $\theta_1' \leq \theta_1$ implies $\theta_1'^* = \theta_1^*$ so the application fPd makes sense.) From safety of h and the assumption $\Pi \cup \Pi_1 \cup \Pi_2 \subseteq P$, we get by induction on e_1 that $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(f)$, and we get $\mathsf{safe}\ \theta_1'(d)$ by induction on e_2. By $\theta_1' \leq \theta_1$ and Fact 4.3 we have $\mathsf{safe}\ \theta_1'(d) \Rightarrow \mathsf{safe}\ \theta_1(d)$. Then by definition $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(f)$ we get $\mathsf{safe}\ \theta_2(fPd)$.
- Case `letrec` $f(x) = e_1$ `in` e_2: Then, $\Pi \cup \Pi_1 \subseteq P$.
  Now $u = [\![\Delta^*, f : \theta_1^* \to \theta_2^* \vdash e_2 : \theta^*]\!] nP[h \mid f \mapsto \mathit{fix}\ G]$, where

  $$G(g) = \lambda P'.\ \lambda d.\ [\![\Delta^*, f : \theta_1^* \to \theta_2^*, x : \theta_1^* \vdash e_1 : \theta_2^*]\!] nP'[h \mid f \mapsto g, x \mapsto d]$$

  To get $\mathsf{safe}\ \theta(u)$ by induction for e_2, we need $\Pi_1 \subseteq P$ and

  $$\mathsf{safe}\ (\Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2)[h \mid f \mapsto \mathit{fix}\ G]$$

  The former follows from the assumption $\Pi \cup \Pi_1 \subseteq P$. The latter follows from assumption, $\mathsf{safe}\ \Delta(h)$, and $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(\mathit{fix}\ G)$. We proceed to show safety of $\mathit{fix}\ G$.
  Now $\mathit{fix}\ G = \bigsqcup_i g_i$, where $g_0 = \lambda P''.\ \lambda d \in [\![\theta_1^*]\!].\ \bot$ and $g_{i+1} = G(g_i)$. And, $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(\mathit{fix}\ G)$ is a consequence of the following claim:

  $$\forall i.\ \mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(g_i) \tag{2}$$

  Then from Lemma 4.4, we get $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(\bigsqcup_i g_i)$. It remains to show (2), for which we proceed by induction on i.
  Base case: Show $\mathsf{safe}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(g_0)$. Assume any $P'' \in \mathscr{P}(\mathsf{Privileges})$ and any $v \in [\![\theta_1^*]\!]$, such that $\Pi \subseteq P''$ and $\mathsf{safe}\ \theta_1(v)$. Then $g_0 P'' v = \bot \neq \star$ and $\mathsf{safe}\ \theta_2(g_0 P'' v)$ holds.

Induction step: Assume safe $(\theta_1 \xrightarrow{\Pi} \theta_2)(g_i)$, to show safe $(\theta_1 \xrightarrow{\Pi} \theta_2)(g_{i+1})$.
Now $g_{i+1} = G(g_i) = \lambda P'.\, \lambda d.\, [\![\Delta^*, f : \theta_1^* \to \theta_2^*, x : \theta_1^* \vdash e_1 : \theta_2^*]\!] n P[h \mid f \mapsto g_i, x \mapsto d]$. Assume any $P'' \in \mathscr{P}(\mathsf{Privileges})$ and $v \in [\![\theta_1^*]\!]$, such that $\Pi \subseteq P''$ and safe $\theta_1(v)$. Then

$$g_{i+1}P''(v) = [\![\Delta^*, f : \theta_1^* \to \theta_2^*, x : \theta_1^* \vdash e_1 : \theta_2^*]\!] n P''[h \mid f \mapsto g_i, x \mapsto v]$$

Note that safe $(\Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2, x : \theta_1)[h \mid f \mapsto g_i, x \mapsto v]$. Therefore, by the main induction hypothesis on the typing derivation $\Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2, x : \theta_1;\ n \vdash e_1 : \theta_2,\ \Pi$, since $\Pi \subseteq P$, we obtain safe $\theta_2(g_{i+1}P''v)$.

- Case `signs` n' e: Then $\Pi \subseteq P$ and $u = [\![\Delta^* \vdash e : \theta^*]\!] n'(P \cap \mathscr{A}(n'))h$. The induction hypothesis on the typing derivation of e can be used to obtain safe $\theta(u)$, because $\Pi \subseteq (P \cap \mathscr{A}(n'))$ which follows from assumption $\Pi \subseteq P$ and side condition $\Pi \subseteq \mathscr{A}(n')$ on the antecedent $\Delta;\ n' \vdash e : \theta,\ \Pi$ of $\Delta;\ n \vdash$ `signs` $n'\ e : \theta,\ \Pi$.
- Case `dopriv` p `in` e: Then $\Pi \subseteq P$ and $u = [\![\Delta^* \vdash e : \theta^*]\!] n(P \sqcup_n \{p\})h$. By the induction hypothesis for e, noting that $(\Pi \sqcup_n \{p\}) \subseteq (P \sqcup_n \{p\})$, we have safe $\theta(u)$.
- Case `check` p `for` e:
  Then $\Pi \cup \{p\} \subseteq P$, hence $p \in P$. Now $u =$ **if** $p \in P$ **then** $[\![\Delta^* \vdash e : \theta^*]\!] nPh$ **else** $\star$. Since $p \in P$, we have, $u = [\![\Delta^* \vdash e : \theta^*]\!] nPh$ and, by the induction hypothesis on the typing derivation of e, we have safe $\theta(u)$.
- Case `test` p `then` e_1 `else` e_2: Then $\Pi_1 \cup \Pi_2 \subseteq P$ and

  $$u = \textbf{if } p \in P \textbf{ then } [\![\Delta^* \vdash e_1 : \theta^*]\!] nPh \textbf{ else } [\![\Delta^* \vdash e_2 : \theta^*]\!] nPh$$

  We have two cases. Suppose $p \in P$. Then, by induction hypothesis on typing derivation of e_1 and noting that $\Pi_1 \subseteq P$, we have $u = [\![\Delta^* \vdash e_1 : \theta^*]\!] nPh$ and safe $\theta(u)$. The case where $p \notin \mathscr{P}$, is symmetric. ■

5 Some program transformations

Using the eager semantics it is straightforward to justify program transformations that can be used for optimization. This section shows how checks can be eliminated from the password example and also considers the proofs of some primitive equations from Fournet and Gordon's work [11].

5.1 Transformations that eliminate checks

First, we list a series of program transformations that move checking of privileges "outwards".

`if` e `then check` p `for` e_1 `else check` p `for` e_2	$=$	`check` p `for if` e `then` e_1 `else` e_2
e_1(`check` p `for` e_2)	$=$	`check` p `for` $e_1\, e_2$
`test` p `then` e_1 `else check` p `for` e_2	$=$	`check` p `for test` p `then` e_1 `else` e_2
`test` p' `then check` p `for` e_1 `else check` p `for` e_2	$=$	`check` p `for test` p' `then` e_1 `else` e_2
`letrec` $f(x) = e_1$ `in check` p `for` e_2	$=$	`check` p `for letrec` $f(x) = e_1$ `in` e_2
`check` p `for check` p `for` e	$=$	`check` p `for` e

These are unconditional equalities, as the reader can verify using the denotational semantics (Figure 2). We emphasize that this means extensional equality of the functions denoted by the two sides. In particular, $e = e'$ means $[\![D \vdash e : t]\!] nPh = [\![D \vdash e' : t]\!] nPh$ for all n, P, h. Corresponding to Fournet and

Gordon one may consider a slightly weaker notion of equality where P ranges over subsets of $\mathscr{A}(n)$, as indicated by (1) in Section 3.2. Later we encounter one transformation that only holds for that weaker equality.

Once checks have been moved outward, some can be eliminated. To eliminate a check, it must be known definitely to succeed, *e.g.*, because it has been enabled for an authorized principal. We give an example transformation of this kind in Theorem 5.4, formulated in terms of the following notions concerning expressions that do not depend on privilege p.

Definition 5.1 (p-purity) An expression e is p-pure if e has no sub-expressions of the form `check` p `for` e' or `test` p `then` e' `else` e''.

For each type t we define semantic p-purity as a predicate $\mathsf{pure}\ p\ t$ on $[\![t]\!]_{\perp\star}$, as follows: $\mathsf{pure}\ p\ t(\perp) \Leftrightarrow \mathsf{true}$ and $\mathsf{pure}\ p\ t(\star) \Leftrightarrow \mathsf{true}$ for all t. For values other than $\perp$ and $\star$, the definition is by induction on structure of t.

$$\begin{array}{lcl} \mathsf{pure}\ p\ \mathtt{bool}(b) & \Leftrightarrow & \mathsf{true} \\ \mathsf{pure}\ p\ (t_1 \to t_2)(f) & \Leftrightarrow & \forall P \in \mathscr{P}(\mathsf{Privileges}).\forall d \in [\![t_1]\!]. \\ & & \mathsf{pure}\ p\ t_1(d) \Rightarrow \mathsf{pure}\ p\ t_2(fPd) \wedge fPd = f(P-\{p\})d \end{array}$$

Finally, for environment $h \in [\![D]\!]$ we define $\mathsf{pure}\ p\ D(h)$ iff $\mathsf{pure}\ p\ t(h.x)$ for all $x : t$ in D.

Lemma 5.2 Suppose $u : \mathbb{N} \to [\![t_1 \to t_2]\!]$ is an ascending chain. Then $\forall i.\mathsf{pure}\ p\ (t_1 \to t_2)(u_i)$ implies $\mathsf{pure}\ p\ (t_1 \to t_2)(\sqcup_i u_i)$.

Proof: By definition of pure and since joins are given pointwise. ■

Lemma 5.3 If e is p-pure and typable as $D \vdash e : t$, then for all n, P, h with $\mathsf{pure}\ p\ D(h)$ we have

$$[\![D \vdash e : t]\!]nPh = [\![D \vdash e : t]\!]n(P-\{p\})h$$

and $\mathsf{pure}\ p\ t([\![D \vdash e : t]\!]nPh)$.

Proof: By induction on e. We observe for any D, n, P, h with $\mathsf{pure}\ p\ D\ (h)$

- Case `true`: The equation is direct from the semantics, which is independent of P. For p-purity of `true`, the result holds by definition of $\mathsf{pure}\ p$ `bool`.
- Case x: the equation is direct from the semantics which is independent of P. For p-purity of $[\![D \vdash x : t]\!]nPh$, the result holds by hypothesis on h.
- Case `if` e_1 `then` e_2 `else` e_3: straightforward use of induction.
- Case `fun` $x.\ e$: The equation holds because the semantics is independent of P. Purity holds by induction on e.
- Case $e_1 e_2$: To show the equation, we use that $[\![D \vdash e_1]\!]$ is p-pure, which holds by induction. To show purity, we again use purity of e_1 as well as purity of e_2.
- Case `letrec` $f(x) = e_1$ `in` e_2: By induction on e_2, using Lemma 5.2.
- Case `signs` n' e: The equation is direct from semantics, using the fact that $(P \cap \mathscr{A}(n')) - \{p\} = (P-\{p\}) \cap \mathscr{A}(n')$.

- Case `dopriv` p' `in` e: We first consider the case where p' is distinct from p. We have

$$\begin{array}{lll} & [\![D \vdash \texttt{dopriv}\ p'\ \texttt{in}\ e : t]\!] nPh & \\ = & [\![D \vdash e : t]\!] n(P \sqcup_n \{p'\})h & \text{semantics} \\ = & [\![D \vdash e : t]\!] n((P \sqcup_n \{p'\}) - \{p\})h & \text{induction hyp.} \\ = & [\![D \vdash e : t]\!] n((P - \{p\} \sqcup_n \{p'\})h & p, p' \text{distinct} \\ = & [\![D \vdash \texttt{dopriv}\ p'\ \texttt{in}\ e : t]\!] n(P - \{p\})h & \text{semantics} \end{array}$$

  In case p' is p we have

$$\begin{array}{lll} & [\![D \vdash \texttt{dopriv}\ p\ \texttt{in}\ e : t]\!] nPh & \\ = & [\![D \vdash e : t]\!] n(P \sqcup_n \{p\})h & \text{semantics} \\ = & [\![D \vdash e : t]\!] n((P - \{p\}) \sqcup_n \{p\})h & \text{see below} \\ = & [\![D \vdash \texttt{dopriv}\ p\ \texttt{in}\ e : t]\!] n(P - \{p\})h & \text{semantics} \end{array}$$

  The middle step is by cases on whether $p \in \mathscr{A}(n)$. If it is, the step holds by simply by definition of $\sqcup_n$. If not, the step holds by induction on e.

- Case `check` p' `for` e: Here p' is distinct from p, by p-purity. We observe

$$\begin{array}{lll} & [\![D \vdash \texttt{check}\ p'\ \texttt{for}\ e : t]\!] nPh & \\ = & \textbf{if } p' \in P \textbf{ then } [\![D \vdash e : t]\!] nPh \textbf{ else } \star & \text{semantics} \\ = & \textbf{if } p' \in P - \{p\} \textbf{ then } [\![D \vdash e : t]\!] n(P - \{p\})h \textbf{ else } \star & p', p \text{ distinct, ind. for } e \\ = & [\![D \vdash \texttt{check}\ p'\ \texttt{for}\ e : t]\!] n(P - \{p\})h & \end{array}$$

- Case `test` p' `then` e_1 `else` e_2: Again, p' is distinct from p, and the argument is similar to `check`.

Theorem 5.4 For all n, all $p \in \mathscr{A}(n)$, and all p-pure closed terms e

$$\texttt{signs}\ n\ \texttt{dopriv}\ p\ \texttt{in}\ \texttt{check}\ p\ \texttt{for}\ e \quad = \quad \texttt{signs}\ n\ e$$

Proof: Let h be the empty environment, for e which is closed. We observe for any n', P:

$$\begin{array}{lll} & [\![\texttt{signs}\ n\ \texttt{dopriv}\ p\ \texttt{in}\ \texttt{check}\ p\ \texttt{for}\ e]\!] n'Ph & \\ = & [\![\texttt{dopriv}\ p\ \texttt{in}\ \texttt{check}\ p\ \texttt{for}\ e]\!] n(\mathscr{A}(n) \cap P)h & \text{semantics} \\ = & [\![\texttt{check}\ p\ \texttt{for}\ e]\!] n((\mathscr{A}(n) \cap P) \sqcup_n \{p\})h & \text{semantics} \\ = & [\![\texttt{check}\ p\ \texttt{for}\ e]\!] n((\mathscr{A}(n) \cap P) \cup \{p\})h & \text{def } \sqcup_n, \text{ using } p \in \mathscr{A}(n) \\ = & [\![e]\!] n((\mathscr{A}(n) \cap P) \cup \{p\})h & \text{semantics} \\ = & [\![e]\!] n(\mathscr{A}(n) \cap P)h & e \text{ and } h \text{ are } p\text{-pure, Lemma 5.3} \\ = & [\![\texttt{signs}\ n\ e]\!] n'Ph & \text{semantics} \end{array}$$

In the penultimate step, two uses are needed for the lemma: to remove p and, in the case that $p \in P$, to add it back. ■

In order to deal with the password example we need the following conditional equivalence.

Theorem 5.5 For all n, all $p \in \mathscr{A}(n)$, and all terms e

$$\texttt{signs}\ n\ \texttt{check}\ p\ \texttt{for}\ e \quad = \quad \texttt{check}\ p\ \texttt{for}\ \texttt{signs}\ n\ e$$

Proof: We observe for any n', P, h:

$$\begin{array}{lll} & [\![\texttt{signs}\ n\ \texttt{check}\ p\ \texttt{for}\ e]\!]n'Ph & \\ = & [\![\texttt{check}\ p\ \texttt{for}\ e]\!]n(P \cap \mathscr{A}(n))h & \text{semantics} \\ = & \textbf{if}\ p \in (P \cap \mathscr{A}(n))\ \textbf{then}\ [\![e]\!]n(P \cap \mathscr{A}(n))h\ \textbf{else}\ \star & \text{semantics} \\ = & \textbf{if}\ p \in P\ \textbf{then}\ [\![e]\!]n(P \cap \mathscr{A}(n))h\ \textbf{else}\ \star & \text{sets, since } p \in \mathscr{A}(n) \\ = & \textbf{if}\ p \in P\ \textbf{then}\ [\![\texttt{signs}\ n\ e]\!]n'Ph\ \textbf{else}\ \star & \text{semantics} \\ = & [\![\texttt{check}\ p\ \texttt{for}\ \texttt{signs}\ n\ e]\!]n'Ph & \text{semantics} \end{array}$$

■

The above proofs are examples of the benefit of a compositional semantics. The proofs are by direct calculation, without need for induction. For Theorem 5.4, the proof goes through for open terms as well, if the environment h is pure. One expects built-in constants to have pure and safe values.

5.2 The password example

We now revisit the password example, using Theorems 5.4 and 5.5 to eliminate checks. We abbreviate $user, root$ as u, r.

$$\begin{array}{ll} & passwd(\text{“mypass”}) \\ = & \{\text{because } passwd = (\texttt{fun}\ x.\ \texttt{signs}\ r\ \texttt{check}\ p\ \texttt{for}\ \texttt{dopriv}\ w\ \texttt{in}\ writepass(x))\} \\ & \texttt{signs}\ r\ \texttt{check}\ p\ \texttt{for}\ \texttt{dopriv}\ w\ \texttt{in}\ writepass(\text{“mypass”}) \\ = & \{\text{because } writepass = (\texttt{fun}\ x.\ \texttt{signs}\ r\ \texttt{check}\ w\ \texttt{for}\ hwWrite(x, \text{“/etc/password”}))\} \\ & \texttt{signs}\ r\ \texttt{check}\ p\ \texttt{for}\ \texttt{dopriv}\ w\ \texttt{in}\ \texttt{signs}\ r\ \texttt{check}\ w\ \texttt{for}\ hwWrite(\text{“mypass”}, \text{“/etc/password”}) \\ = & \{\text{by Theorem 5.5 since } \mathscr{A}(r) = \{p, w\}\} \\ & \texttt{check}\ p\ \texttt{for}\ \texttt{signs}\ r\ \texttt{dopriv}\ w\ \texttt{in}\ \texttt{check}\ w\ \texttt{for}\ \texttt{signs}\ r\ hwWrite(\text{“mypass”}, \text{“/etc/password”}) \\ = & \{\text{by Theorem 5.4 since } w \in \mathscr{A}(r) \text{ and } \texttt{signs}\ r\ hwWrite(\ldots) \text{ is } p\text{-pure closed}\} \\ & \texttt{check}\ p\ \texttt{for}\ \texttt{signs}\ r\ \texttt{signs}\ r\ hwWrite(\text{“mypass”}, \text{“/etc/password”}) \\ = & \{\text{because } \texttt{signs}\ n\ \texttt{signs}\ n\ e = \texttt{signs}\ n\ e\} \\ & \texttt{check}\ p\ \texttt{for}\ \texttt{signs}\ r\ hwWrite(\text{“mypass”}, \text{“/etc/password”}) \end{array}$$

In the last step we used the unconditional equation $\texttt{signs}\ n\ \texttt{signs}\ n\ e = \texttt{signs}\ n\ e$ which is easily proved. Finally, we obtain:

$$\begin{array}{ll} & use = \texttt{signs}\ u\ \texttt{dopriv}\ p\ \texttt{in}\ passwd(\text{“mypass”}) \\ = & \texttt{signs}\ u\ \texttt{dopriv}\ p\ \texttt{in}\ \texttt{check}\ p\ \texttt{for}\ \texttt{signs}\ r\ hwWrite(\text{“mypass”}, \text{“/etc/password”}) \\ = & \{\text{by Theorem 5.4 since } p \in \mathscr{A}(u) \text{ and } \texttt{signs}\ r\ hwWrite(\ldots) \text{ is } p\text{-pure closed}\} \\ & \texttt{signs}\ u\ \texttt{signs}\ r\ hwWrite(\text{“mypass”}, \text{“/etc/password”}) \end{array}$$

5.3 Other transformations

We now give an example of a transformation which employs a weaker notion of equality than the ones in Section 5.1, cf. (1). This is the TestGrant equation of Fournet and Gordon,[5] which, in our notation amounts to proving, for all n, P, h, p, e_1, e_2, with $P \subseteq \mathscr{A}(n)$,

$$[\![\texttt{test}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]nPh = [\![\texttt{test}\ p\ \texttt{then}\ \texttt{dopriv}\ p\ \texttt{in}\ e_1\ \texttt{else}\ e_2]\!]nPh \quad (3)$$

[5] Theirs is slightly more general, as their `test` and `dopriv` apply to permission sets; this can be desugared to singleton permissions as in our notation.

To show (3) we use the denotational semantics and prove

$$\textbf{if } p \in P \textbf{ then } [\![e_1]\!]nPh \textbf{ else } [\![e_2]\!]nPh \quad = \quad \textbf{if } p \in P \textbf{ then } [\![\texttt{dopriv } p \texttt{ in } e_1]\!]nPh \textbf{ else } [\![e_2]\!]nPh$$

It suffices to prove that when $p \in P$, we have $[\![e_1]\!]nPh = [\![\texttt{dopriv } p \texttt{ in } e_1]\!]nPh$. We calculate:

$$\begin{array}{lll} & [\![\texttt{dopriv } p \texttt{ in } e_1]\!]nPh & \\ = & [\![e_1]\!]n(P \sqcup_n \{p\})h & \text{semantics} \\ = & [\![e_1]\!]nPh & \text{because } p \in P \text{ and } P \subseteq \mathscr{A}(n) \text{ implies } p \in \mathscr{A}(n) \end{array}$$

We have proved the correctness of several other primitive equations in Fournet and Gordon's paper [11, Section 4.1].[6] Specifically, Frame Frame, Frame Frame Frame, Frame Frame Grant, Frame Grant, Frame Grant Frame, Frame Grant Test, Frame Test Then, Grant Grant, Grant Frame, Grant Frame Grant and Test $\cup$. We have also proved the correctness of their derived equations Frame Appl, Frame Frame Intersect and Frame Grant Intersect. Our proofs of these equations do not require the restriction $P \subseteq \mathscr{A}(n)$.

We have also proved the tail call elimination laws in [11, Section 5.2]. The basic idea in tail call elimination is to not build a new frame for the last call of a function; instead the callee can directly return to the caller's caller. Tail call elimination is problematic with stack inspection, as a stack frame holds the principal for the current code (or, equivalently, the principal's static permissions). As noted earlier, Clements and Felleisen [10] give an abstract machine for which tail call elimination is sound and efficient. The calculus and small-step semantics of Fournet and Gordon [11, Section 5.2] allows a limited modeling of tail call elimination. Here is one of their two transformations, in our notation:

$$\texttt{signs } n_2 \; ((\texttt{fun } x.\; \texttt{signs } n_1 \; e_1) \; e_2) = ((\texttt{fun } x.\; \texttt{signs } n_1 \; e_1) \; e_2) \tag{4}$$

From left to right this can be read as dropping the "frame" of the calling context. In their setting it can be read as a transition, provided that e_2 is a value. They show that this added transition is admissible, in the sense of not changing outcomes, provided that the callee's static permissions are among the caller's, *i.e.* $\mathscr{A}(n_1) \subseteq \mathscr{A}(n_2)$. For our purposes, a value is a boolean literal, a variable (whose value is thus in the environment) or an abstraction. We shall prove that the equation holds in our semantics, under these conditions. For the proof, it is convenient to use the following easily proved fact which holds for any D, e_1, e_2, n, P, h.

$$[\![D \vdash (\texttt{fun } x.\; e_1) \; e_2]\!]nPh \;=\; [\![D, x : t \vdash e_1]\!]nP[h \mid x \mapsto [\![D \vdash e_2]\!]nPh] \tag{5}$$

Note: if e_2 is a value then its semantics $[\![D \vdash e_2]\!]nPh$ is independent from P (though not necessarily from n or h); see Fig. 2. To prove (4) we observe

$$\begin{array}{lll} & [\![D \vdash \texttt{signs } n_2 \; ((\texttt{fun } x.\; \texttt{signs } n_1 \; e_1) \; e_2)]\!]nPh & \\ = & [\![D \vdash ((\texttt{fun } x.\; \texttt{signs } n_1 \; e_1) \; e_2)]\!]n(P \cap \mathscr{A}(n_2))h & \text{by semantics of signs} \\ = & [\![D, x : t \vdash \texttt{signs } n_1 \; e_1]\!]n(P \cap \mathscr{A}(n_2))[h \mid x \mapsto [\![D \vdash e_2]\!]n(P \cap \mathscr{A}(n_2))h] & \text{lemma (5)} \\ = & [\![D, x : t \vdash e_1]\!]n(P \cap \mathscr{A}(n_1))[h \mid x \mapsto [\![D \vdash e_2]\!]n(P \cap \mathscr{A}(n_2))h] & \text{sem., } \mathscr{A}(n_1) \subseteq \mathscr{A}(n_2) \\ = & [\![D, x : t \vdash e_1]\!]n(P \cap \mathscr{A}(n_1))[h \mid x \mapsto [\![D \vdash e_2]\!]nPh] & e_2 \text{ value, Note above} \\ = & [\![D, x : t \vdash \texttt{signs } n_1 \; e_1]\!]nP[h \mid x \mapsto [\![D \vdash e_2]\!]nPh] & \text{semantics of signs} \\ = & [\![D \vdash ((\texttt{fun } x.\; \texttt{signs } n_1 \; e_1) \; e_2)]\!]nPh & \text{lemma (5)} \end{array}$$

As with most of the other equations, the restriction $P \subseteq \mathscr{A}(n)$ is not necessary here. Fournet and Gordon give a second equation, also provable in our setting, that models tail calls involving `dopriv`.

[6] For a few we need to consider the evident generalization of our language that allows permission sets in `dopriv` (for Grant Grant, Grant Frame, and Frame Grant Intersect) and in `test` (for Test $\cup$).

6 Using the Static Analysis

Section 5 gives several program transformations that can be justified by the eager denotational semantics of our language. A more drastic transformation is possible under some conditions. The safety results of Section 4 show that if the static analysis derives a judgement $\Delta;\ n \vdash e : \theta, \Pi$, then executing e using a privilege set that contains at least the enabled privileges Π would not lead to a security error. We should therefore be able to drop all `dopriv`'s and `check`'s from e. If e is `test`-free, we can then show that the meaning of e is the same as its meaning with `dopriv`'s and `check`'s erased. This is formalized below.

Definition 6.1 The erasure translation $(.)^-$ is defined as follows:

$$
\begin{array}{lcl}
\texttt{true}^- & = & \texttt{true} \\
x^- & = & x \\
(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3)^- & = & \texttt{if } e_1^- \texttt{ then } e_2^- \texttt{ else } e_3^- \\
(\texttt{fun } x.\ e)^- & = & \texttt{fun } x.\ e^- \\
(\texttt{letrec } f(x) = e_1 \texttt{ in } e_2)^- & = & \texttt{letrec } f(x) = e_1^- \texttt{ in } e_2^- \\
(\texttt{signs } n\ e)^- & = & \texttt{signs } n\ e^- \\
(\texttt{dopriv } p \texttt{ in } e)^- & = & e^- \\
(\texttt{check } p \texttt{ for } e)^- & = & e^- \\
(\texttt{test } p \texttt{ then } e_1 \texttt{ else } e_2)^- & & \text{is undefined.}
\end{array}
$$

Theorem 6.2 Let e be `test`-free and let $\varnothing;\ n \vdash e : \texttt{bool}, \Pi$. Then for all $P \in \mathscr{P}(\mathsf{Privileges})$, if $\Pi \subseteq P$ then $[\![\varnothing \vdash e : \texttt{bool}]\!] n P\{\} = [\![\varnothing \vdash e^- : \texttt{bool}]\!] n P\{\}$.

Proof: Immediate consequence of Lemma 6.6 and definition rel `bool` below. ■

Definition 6.3 For each annotated type θ the relation rel θ on $[\![\theta^*]\!]_{\perp\star}$ is defined as follows: For all θ, rel $\theta\ \perp\ \perp$ holds and otherwise rel $\theta\ d\ d'$ is false if either d or d' is in $\{\perp, \star\}$. For values other than $\perp, \star$, the definition is by induction on structure of θ.

$$
\begin{array}{rcl}
\mathsf{rel}\ \texttt{bool}\ b\ b' & \Leftrightarrow & b = b' \\
\mathsf{rel}\ (\theta_1 \xrightarrow{\Pi} \theta_2)\ f\ f' & \Leftrightarrow & \forall P \in \mathscr{P}(\mathsf{Privileges}).\forall d, d' \in [\![\theta_1^*]\!]. \\
 & & \Pi \subseteq P \wedge \mathsf{rel}\ \theta_1\ d\ d' \Rightarrow \mathsf{rel}\ \theta_2\ (fPd)\ (f'Pd')
\end{array}
$$

For annotated type environment Δ, the predicate rel Δ on $[\![\Delta^*]\!]$ is defined by rel $\Delta\ h\ h' \Leftrightarrow \mathit{dom}(h) = \mathit{dom}(h')$ and $\forall x \in \mathit{dom}(h).\mathsf{rel}\ (\Delta.x)\ (h.x)\ (h'.x)$.

Fact 6.4 $\theta \leq \theta'$ and rel $\theta\ d\ d'$ imply rel $\theta'\ d\ d'$.

Proof: By induction on derivation of $\theta \leq \theta'$. The result is clear for $\texttt{bool} \leq \texttt{bool}$. For $(\theta_1 \xrightarrow{\Pi} \theta_2) \leq (\theta_1' \xrightarrow{\Pi'} \theta_2')$, assume rel $(\theta_1 \xrightarrow{\Pi} \theta_2)\ f\ f'$. To show rel $(\theta_1' \xrightarrow{\Pi'} \theta_2')\ f\ f'$, consider any $P \in \mathscr{P}(\mathsf{Privileges})$, such that $\Pi' \subseteq P$, and any $d, d' \in [\![\theta_1'^*]\!]$ with rel $\theta_1'\ d\ d'$. From the subtyping, we know that $\Pi \subseteq \Pi'$, hence $\Pi \subseteq P$. Moreover, by induction on derivation of $\theta_1' \leq \theta_1$, we obtain rel $\theta_1'\ d\ d'$ implies rel $\theta_1\ d\ d'$. Hence from assumption rel $(\theta_1 \xrightarrow{\Pi} \theta_2)\ f\ f'$, we obtain rel $\theta_2(fPd)(f'Pd')$. Now by induction on derivation $\theta_2 \leq \theta_2'$, we obtain rel $\theta_2'(fPd)(f'Pd')$. ■

Fact 6.5 The relation rel preserves lubs. That is, for any θ, let $u, u' : \mathbb{N} \to [\![\theta^*]\!]_{\perp\star}$ be ascending chains. Then, $\forall i.\mathsf{rel}\ \theta\ u_i\ u'_i$ implies $\mathsf{rel}\ \theta\ (\bigsqcup_i u_i)(\bigsqcup_i u'_i)$.

Proof: By structural induction on θ. When $\theta = \mathtt{bool}$, we have $\bigsqcup_i u_i = \bigsqcup_i u'_i = \mathsf{true}$ or false or $\perp$. Thus the result holds by definition rel.
When $\theta = (\theta_1 \xrightarrow{\Pi} \theta_2)$, assume $P \in \mathscr{P}(\mathsf{Privileges})$ and $d, d' \in [\![\theta_1^*]\!]$, such that $\Pi \subseteq P$ and $\mathsf{rel}\ \theta_1 d\ d'$. Then, from assumption $\mathsf{rel}\ (\theta_1 \xrightarrow{\Pi} \theta_2)\ u_i\ u'_i$ we obtain $\mathsf{rel}\ \theta_2\ (u_i P d)(u'_i P d')$ for every i. Hence, by the induction hypothesis on θ_2, we get $\mathsf{rel}\ \theta_2\ (\bigsqcup_i (u_i P d))(\bigsqcup_i (u'_i P d'))$. Because lubs are pointwise, we get $\mathsf{rel}\ \theta_2\ ((\bigsqcup_i u_i) P d)((\bigsqcup_i u'_i) P d')$. ■

Lemma 6.6 Suppose $\Delta;\ n \vdash e : \theta,\ \Pi$ is derivable and e is `test`-free. Then for all $P \in \mathscr{P}(\mathsf{Privileges})$, for all $h, h^- \in [\![\Delta^*]\!]$, if $\mathsf{rel}\ \Delta\ h\ h^-$ and $\Pi \subseteq P$ then $\mathsf{rel}\ \theta\ u\ u^-$, where $u = [\![\Delta^* \vdash e : \theta^*]\!] n P h$ and $u^- = [\![\Delta^* \vdash e^- : \theta^*]\!] n P h^-$.

(Note that h^-, u^- are just suggestively named identifiers whereas e^- is the erasure of e.) Theorem 6.2 follows from the lemma because $\mathsf{rel}\ \varnothing\ \{\}\ \{\}$ and by definition $\mathsf{rel}\ \mathtt{bool}$, $[\![\varnothing \vdash e : \mathtt{bool}]\!] n P \{\} = [\![\varnothing \vdash e^- : \mathtt{bool}]\!] n P \{\}$.

Proof: of Lemma. Go by induction on the typing derivation, $\Delta;\ n \vdash e : \theta,\ \Pi$. Throughout, we assume $P \in \mathscr{P}(\mathsf{Privileges})$ and $h, h^- \in [\![\Delta^*]\!]$ and $\mathsf{rel}\ \Delta\ h\ h^-$. Let $u = [\![\Delta^* \vdash e : \theta^*]\!] n P h$ and $u^- = [\![\Delta^* \vdash e^- : \theta^*]\!] n P h^-$ for each case of e.

- Case `true`: Then, $u = \mathsf{true} = u^-$ and $\mathsf{rel}\ \mathtt{bool}\ u\ u^-$ by definition rel.
- Case x: Then, $u = h.x$ and $u^- = h^-.x$. And, $\mathsf{rel}\ \theta\ u\ u^-$ follows from assumption $\mathsf{rel}\ \Delta\ h\ h^-$.
- Case `if` e `then` e_1 `else` e_2: Then $\Pi_1 \cup \Pi_2 \cup \Pi_3 \subseteq P$, and

$$\begin{array}{rcl} u & = & \textbf{if } b \textbf{ then } [\![\Delta^* \vdash e_1 : \theta^*]\!] n P h \textbf{ else } [\![\Delta^* \vdash e_2 : \theta^*]\!] n P h \\ u^- & = & \textbf{if } b^- \textbf{ then } [\![\Delta^* \vdash e_1^- : \theta^*]\!] n P h^- \textbf{ else } [\![\Delta^* \vdash e_2^- : \theta^*]\!] n P h^- \end{array}$$

  where $b = [\![\Delta^* \vdash e : \mathtt{bool}]\!] n P h$ and $b^- = [\![\Delta^* \vdash e^- : \mathtt{bool}]\!] n P h^-$. By the induction hypothesis on the typing derivation of e, noting that $\Pi_1 \subseteq P$, we have $\mathsf{rel}\ \mathtt{bool}\ b\ b^-$. If $b = \perp = b^-$ then $u = \perp = u^-$ and $\mathsf{rel}\ \theta\ \perp\ \perp$. Otherwise, $b = \mathsf{true}$ or $b = \mathsf{false}$. In the former case, by the induction hypothesis on the typing derivation of e_1, noting that $\Pi_2 \subseteq P$, we have $\mathsf{rel}\ \theta\ u\ u^-$. In the latter case, by the induction hypothesis on the typing derivation of e_2, noting that $\Pi_3 \subseteq P$, we have $\mathsf{rel}\ \theta\ u\ u^-$.

- Case `fun` $x.\ e$: Then $\begin{array}{rcl} u & = & \lambda P'.\, \lambda d.\, [\![\Delta^*, x : \theta_1^* \vdash e : \theta_2^*]\!] n P' [h \mid x \mapsto d] \\ u^- & = & \lambda P'.\, \lambda d^-.\, [\![\Delta^*, x : \theta_1^* \vdash e^- : \theta_2^*]\!] n P' [h^- \mid x \mapsto d^-] \end{array}$

  To prove $\mathsf{rel}\ (\theta_1 \xrightarrow{\Pi} \theta_2)\ u\ u^-$, consider any $P' \in \mathscr{P}(\mathsf{Privileges})$ and any $d, d^- \in [\![\theta_1^*]\!]$ such that $\Pi \subseteq P'$ and $\mathsf{rel}\ \theta_1\ d\ d^-$, to show $\mathsf{rel}\ \theta_2\ (u P' d')\ (u^- P' d^-)$. By semantics,

$$\begin{array}{rcl} u P' d & = & [\![\Delta^*, x : \theta_1^* \vdash e : \theta_2^*]\!] n P' [h \mid x \mapsto d] \\ u^- P' d^- & = & [\![\Delta^*, x : \theta_1^* \vdash e^- : \theta_2^*]\!] n P' [h^- \mid x \mapsto d^-] \end{array}$$

  So the induction hypothesis for e yields $\mathsf{rel}\ \theta_2\ (u P' d)\ (u^- P' d^-)$ provided that $\Pi \subseteq P'$ and $\mathsf{rel}\ (\Delta, x : \theta_1)[h \mid x \mapsto d][h^- \mid x \mapsto d^-]$. We have $\Pi \subseteq P'$ by assumption, and $\mathsf{rel}\ (\Delta, x : \theta_1)[h \mid x \mapsto d][h^- \mid x \mapsto d^-]$ follows from $\mathsf{rel}\ \Delta\ h\ h^-$ and $\mathsf{rel}\ \theta_1\ d\ d^-$.

- Case $e_1\ e_2$: Let $f = [\![\Delta^* \vdash e_1 : (\theta_1 \xrightarrow{\Pi} \theta_2)^*]\!] nPh$ and $d = [\![\Delta^* \vdash e_2 : \theta_1'^*]\!] nPh$, so that $u = fPd$. Let $f^- = [\![\Delta^* \vdash e_1^- : (\theta_1 \xrightarrow{\Pi} \theta_2)^*]\!] nPh^-$ and $d^- = [\![\Delta^* \vdash e_2^- : \theta_1'^*]\!] nPh^-$, so that $u^- = f^- P d^-$. (Recall that $\theta_1' \leq \theta_1$ implies $\theta_1'^* = \theta_1^*$ so the applications fPd and $f^- P d^-$ make sense.) From $\mathsf{rel}\ \Delta\ h\ h^-$ and assumption $\Pi \cup \Pi_1 \cup \Pi_2 \subseteq P$, we get by induction on e_1 that $\mathsf{rel}\ (\theta_1 \xrightarrow{\Pi} \theta_2)\ f\ f^-$, and we get $\mathsf{rel}\ \theta_1'\ d\ d^-$ by induction on e_2. By $\theta_1' \leq \theta_1$ and Fact 6.4 we have $\mathsf{rel}\ \theta_1'\ d\ d^- \Rightarrow \mathsf{rel}\ \theta_1\ d\ d^-$. Then by definition $\mathsf{rel}\ (\theta_1 \xrightarrow{\Pi} \theta_2)\ f\ f^-$, since $\Pi \subseteq P$, we get $\mathsf{rel}\ \theta_2 (fPd)(f^- P d^-)$.
- Case $\texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e_2$: Then, $\Pi \cup \Pi_1 \subseteq P$.
  Now $u = [\![\Delta^*, f : (\theta_1 \xrightarrow{\Pi} \theta_2)^* \vdash e_2 : \theta^*]\!] nP[h \mid f \mapsto \mathit{fix}\ G]$
  $u^- = [\![\Delta^*, f : (\theta_1 \xrightarrow{\Pi} \theta_2)^* \vdash e_2^- : \theta^*]\!] nP[h^- \mid f \mapsto \mathit{fix}\ G^-]$
  where $G(g) = \lambda P'.\, \lambda d.\, [\![\Delta^*, f : (\theta_1 \xrightarrow{\Pi} \theta_2)^*, x : \theta_1^* \vdash e_1 : \theta_2^*]\!] nP'[h \mid f \mapsto g, x \mapsto d]$
  $G^-(g^-) = \lambda P'.\, \lambda d^-.\, [\![\Delta^*, f : (\theta_1 \xrightarrow{\Pi} \theta_2)^*, x : \theta_1^* \vdash e_1^- : \theta_2^*]\!] nP'[h^- \mid f \mapsto g^-, x \mapsto d^-]$
  To show $\mathsf{rel}\ \theta\ u\ u^-$ by induction on e_2, we need $\Pi_1 \subseteq P$ and

$$\mathsf{rel}\ (\Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2)\ [h \mid f \mapsto \mathit{fix}\ G]\ [h^- \mid f \mapsto \mathit{fix}\ G^-]$$

  The former follows from assumption $\Pi \cup \Pi_1 \subseteq P$. The latter follows from assumption, $\mathsf{rel}\ \Delta\ h\ h^-$, and $\mathsf{rel}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(\mathit{fix}\ G)(\mathit{fix}\ G^-)$, which we now proceed to show.
  Now $\mathit{fix}\ G = \bigsqcup_i g_i$, where $g_0 = \lambda P'.\, \lambda d \in [\![\theta_1^*]\!].\, \bot$ and $g_{i+1} = G(g_i)$. Also $\mathit{fix}\ G^- = \bigsqcup_i g_i^-$, where $g_0^- = \lambda P'.\, \lambda d^- \in \theta_1^*.\, \bot$ and $g_{i+1}^- = G^-(g_i^-)$. And, $\mathsf{rel}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(\mathit{fix}\ G)(\mathit{fix}\ G^-)$ is a consequence of the following claim:

$$\forall i.\ \mathsf{rel}\ (\theta_1 \xrightarrow{\Pi} \theta_2)\ g_i\ g_i^- \tag{6}$$

  Then from Lemma 6.5, we get $\mathsf{rel}\ (\theta_1 \xrightarrow{\Pi} \theta_2)(\bigsqcup_i g_i)(\bigsqcup_i g_i^-)$. It remains to show (6), for which we proceed by induction on i.
  Base case: Show $\mathsf{rel}\ (\theta_1 \xrightarrow{\Pi} \theta_2)\ g_0\ g_0^-$. Assume any $P' \in \mathscr{P}(\mathsf{Privileges})$ and any $v, v^- \in [\![\theta_1^*]\!]$, such that $\Pi \subseteq P'$ and $\mathsf{rel}\ \theta_1\ v\ v^-$. Then $g_0 P' v = \bot = g_0^- P' v^-$ and $\mathsf{rel}\ \theta_2 (g_0 P' v)(g_0^- P' v^-)$.
  Induction step: Assume $\mathsf{rel}\ (\theta_1 \xrightarrow{\Pi} \theta_2)\ g_i\ g_i^-$, to show $\mathsf{rel}\ (\theta_1 \xrightarrow{\Pi} \theta_2)\ g_{i+1}\ g_{i+1}^-$.
  Now $g_{i+1} = \lambda P'.\, \lambda d.\, [\![\Delta^*, f : (\theta_1 \xrightarrow{\Pi} \theta_2)^*, x : \theta_1^* \vdash e_1 : \theta_2^*]\!] nP[h \mid f \mapsto g_i, x \mapsto d]$
  $g_{i+1}^- = \lambda P'.\, \lambda d^-.\, [\![\Delta^*, f : (\theta_1 \xrightarrow{\Pi} \theta_2)^*, x : \theta_1^* \vdash e_1^- : \theta_2^*]\!] nP[h^- \mid f \mapsto g_i^-, x \mapsto d^-]$
  Assume any $P' \in \mathscr{P}(\mathsf{Privileges})$ and $v, v^- \in [\![\theta_1^*]\!]$, such that $\Pi \subseteq P'$ and $\mathsf{rel}\ \theta_1\ v\ v^-$. Then

$$\begin{aligned} g_{i+1} P' v &= [\![\Delta^*, f : (\theta_1 \xrightarrow{\Pi} \theta_2)^*, x : \theta_1^* \vdash e_1 : \theta_2^*]\!] nP'[h \mid f \mapsto g_i, x \mapsto v] \\ g_{i+1}^- P' v^- &= [\![\Delta^*, f : (\theta_1 \xrightarrow{\Pi} \theta_2)^*, x : \theta_1^* \vdash e_1 : \theta_2^*]\!] nP'[h^- \mid f \mapsto g_i^-, x \mapsto v^-] \end{aligned}$$

  Note that $\mathsf{rel}\ (\Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2, x : \theta_1)\ [h \mid f \mapsto g_i, x \mapsto v]\ [h^- \mid f \mapsto g_i^-, x \mapsto v^-]$. Therefore, by the main induction hypothesis on the typing derivation $\Delta, f : \theta_1 \xrightarrow{\Pi} \theta_2, x : \theta_1;\ n \vdash e_1 : \theta_2,\ \Pi$, since $\Pi \subseteq P$, we obtain $\mathsf{rel}\ \theta_2 (g_{i+1} P' v)(g_{i+1}^- P' v^-)$.
- Case $\texttt{signs}\ n'\ e$: Then $\Pi \subseteq P$ and $u = [\![\Delta^* \vdash e : \theta^*]\!] n'(P \cap \mathscr{A}(n'))h$. The induction hypothesis on the typing derivation of e can be used to obtain $\mathsf{rel}\ \theta\ u\ u^-$, because $\Pi \subseteq (P \cap \mathscr{A}(n'))$ which follows from assumption $\Pi \subseteq P$ and side condition $\Pi \subseteq \mathscr{A}(n')$.
- Case $\texttt{dopriv}\ p\ \texttt{in}\ e$: Then $\Pi \subseteq P$ and $u = [\![\Delta^* \vdash e : \theta^*]\!] n(P \sqcup_n \{p\})h$. By the induction hypothesis for e, noting that $(\Pi \sqcup_n \{p\}) \subseteq (P \sqcup_n \{p\})$, we have $\mathsf{rel}\ \theta\ u\ [\![\Delta^* \vdash e^- : \theta^*]\!] n(P \sqcup_n \{p\})h^-$. But now e^- is p-pure. So by Lemma 5.3, $[\![\Delta^* \vdash e^- : \theta^*]\!] n(P \sqcup_n \{p\})h^- = [\![\Delta^* \vdash e^- : \theta^*]\!] nPh^-$. But $u^- = [\![\Delta^* \vdash e^- : \theta^*]\!] nPh^-$. Hence $\mathsf{rel}\ \theta\ u\ u^-$.

- Case `check` p `for` e:
  Then $\Pi \cup \{p\} \subseteq P$, hence $p \in P$. Now $u =$ **if** $p \in P$ **then** $[\![\Delta^* \vdash e : \theta^*]\!] nPh$ **else** $\star$. Since $p \in P$, we have, $u = [\![\Delta^* \vdash e : \theta^*]\!] nPh$ and, by the induction hypothesis on the typing derivation of e, we have $\mathsf{rel}\ \theta\ u\ [\![\Delta^* \vdash e^- : \theta^*]\!] nPh^-$. Hence $\mathsf{rel}\ \theta\ u\ u^-$.

7 Stack Semantics

This section gives a formal semantics using stack inspection, and shows that for standard expressions it coincides with the eager semantics. The connection is much more direct than that of Wallach, Appel and Felten, so a complete detailed proof is not very lengthy.

Because the operations on the stack are in fact stack-like, it is straightforward to give a denotational style semantics parameterized on the stack. We define $\mathsf{Stacks} = \mathsf{nonempty\ list\ of}(\mathsf{Principals} \times \mathscr{P}(\mathsf{Privileges}))$, taken as a cpo ordered by equality. The top is the head of the list, and we write infix :: for cons, so $\langle n, P\rangle :: S$ is the stack with $\langle n, P\rangle$ on top of S, as in Section 2. We also use the predicate chk defined there, and recall the definition $p \in \mathsf{privs}\ S \Leftrightarrow \mathsf{chk}(p, S)$.

Fact 7.1 For all S and all n we have $\mathsf{privs}(S) \cap \mathscr{A}(n) = \mathsf{privs}(\langle n, \varnothing\rangle :: S)$.

Proof: The sets are equal because for any p

$$\begin{array}{rcll} p \in \mathsf{privs}(\langle n, \varnothing\rangle :: S) & \Leftrightarrow & \mathsf{chk}(p, (\langle n, \varnothing\rangle :: S)) & \text{by def privs} \\ & \Leftrightarrow & p \in \mathscr{A}(n) \wedge \mathsf{chk}(p, S) & \text{by def chk and } p \notin \varnothing \\ & \Leftrightarrow & p \in \mathscr{A}(n) \wedge p \in \mathsf{privs}(S) & \text{by def privs} \ \blacksquare \end{array}$$

The stack semantics of an expression is a function

$$([\![D \vdash e : t]\!]) \in \mathsf{Stacks} \to ([\![D]\!]) \to ([\![t]\!])_{\perp\star}$$

Just as in the eager semantics, we need to account for dynamic binding of privileges by interpreting arrow types using an extra parameter. The stack semantics of types is as follows.

$$\begin{array}{rcl} ([\![\mathtt{bool}]\!]) & = & \{\mathsf{true}, \mathsf{false}\} \\ ([\![t_1 \to t_2]\!]) & = & \mathsf{Stacks} \to ([\![t_1]\!]) \to ([\![t_2]\!])_{\perp\star} \end{array}$$

The semantics of expressions is in Figure 4.

We can now relate the denotational semantics of Figure 2 to the stack semantics of Figure 4.

Theorem 7.2 (Consistency) For any standard expression e and stack $(\langle n, P'\rangle :: S)$, we have

$$[\![\varnothing \vdash e : \mathtt{bool}]\!] nP\{\} \ = \ ([\![\varnothing \vdash e : \mathtt{bool}]\!])(\langle n, P'\rangle :: S)\{\} \qquad \text{where } P = \mathsf{privs}(\langle n, P'\rangle :: S).$$

Proof: Immediate consequence of Lemma 7.5 and definition sim `bool` below. ■

As in the proof of safety, we need to generalize the result to allow nonempty contexts. We also consider expressions of arrow type, for which a logical relation is needed.

Definition 7.3 Define data-type indexed family $\mathsf{sim}\ t \subseteq [\![t]\!]_{\perp\star} \times ([\![t]\!])_{\perp\star}$ as follows. For any t, $\mathsf{sim}\ t\ d\ d'$ is true if $d = d'$ and $d \in \{\perp, \star\}$; it is false if $d \neq d'$ and d or d' is in $\{\perp, \star\}$. Otherwise:

$$\begin{array}{rcl} \mathsf{sim}\ \mathtt{bool}\ b\ b' & \Leftrightarrow & b = b' \\ \mathsf{sim}\ (t_1 \to t_2)\ f\ f' & \Leftrightarrow & \forall S \in \mathsf{Stacks}. \forall d \in [\![t_1]\!]. \forall d' \in ([\![t_1]\!]). \\ & & \mathsf{sim}\ t_1\ d\ d' \Rightarrow \mathsf{sim}\ t_2\ (f\ (\mathsf{privs}\ S)\ d)\ (f'\ S\ d') \end{array}$$

$$
\begin{array}{lcl}
(\![D \vdash \texttt{true} : \texttt{bool}]\!)Sh & = & \mathsf{true} \\
(\![D \vdash x : t]\!)Sh & = & h.x \\
(\![D \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t]\!)Sh & = & \textbf{let}\ b\ =\ (\![D \vdash e : \texttt{bool}]\!)Sh\ \textbf{in} \\
 & & \quad \textbf{if}\ b\ \textbf{then}\ (\![D \vdash e_1 : t]\!)Sh\ \textbf{else}\ (\![D \vdash e_2 : t]\!)Sh \\
(\![D \vdash \texttt{fun}\ x.\ e : t_1 \to t_2]\!)Sh & = & \lambda S' \in \mathsf{Stacks}.\ \lambda d \in (\![t_1]\!).\ \\
 & & (\![D, x : t_1 \vdash e : t_2]\!)S'[h \mid x \mapsto d] \\
(\![D \vdash e_1\ e_2 : t_2]\!)Sh & = & \textbf{let}\ f\ =\ (\![D \vdash e_1 : t_1 \to t_2]\!)Sh\ \textbf{in} \\
 & & \textbf{let}\ d = (\![D \vdash e_2 : t_1]\!)Sh\ \textbf{in}\ fSd \\
(\![D \vdash \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e_2 : t]\!)Sh & & \\
\quad = \ \textbf{let}\ G(g)\ =\ \lambda S'.\ \lambda d.\ (\![D, f : t_1 \to t_2, x : t_1 \vdash e_1 : t_2]\!)S[h \mid f \mapsto g, x \mapsto d]\ \textbf{in} & & \\
\quad\quad (\![D, f : t_1 \to t_2 \vdash e_2 : t]\!)S[h \mid f \mapsto \mathit{fix}\ G] & & \\
(\![D \vdash \texttt{signs}\ n'\ e : t]\!)Sh & = & (\![D \vdash e : t]\!)(\langle n', \varnothing\rangle :: S)h \\
(\![D \vdash \texttt{dopriv}\ p\ \texttt{in}\ e : t]\!)(\langle n, P\rangle :: S)h & = & (\![D \vdash e : t]\!)(\langle n, P \cup \{p\}\rangle :: S)h \\
(\![D \vdash \texttt{check}\ p\ \texttt{for}\ e : t]\!)Sh & = & \textbf{if}\ \mathsf{chk}(p, S)\ \textbf{then}\ (\![D \vdash e : t]\!)Sh\ \textbf{else}\ \star \\
(\![D \vdash \texttt{test}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t]\!)Sh & = & \textbf{if}\ \mathsf{chk}(p, S)\ \textbf{then}\ (\![D \vdash e_1 : t]\!)Sh\ \textbf{else}\ (\![D \vdash e_2 : t]\!)Sh
\end{array}
$$

Figure 4: Stack semantics

An environment $h \in [\![D]\!]$ simulates an environment $h' \in (\![D]\!)$, written $\mathsf{sim}\ D\ h\ h'$, provided $\mathsf{sim}\ (D.x)\ (h.x)\ (h'.x)$ for all $x \in \mathit{dom}(h)$.

Lemma 7.4 The relation sim preserves lubs. That is, for any t, if $u : \mathbb{N} \to [\![t]\!]$ and $u' : \mathbb{N} \to (\![t]\!)$ are ascending chains and $\forall i.\mathsf{sim}\ t\ u_i\ u'_i$ then $\mathsf{sim}\ t\ (\bigsqcup_i u_i)\ (\bigsqcup_i u'_i)$.

Proof: Go by structural induction on t. Assume that $\mathsf{sim}\ t\ u_i\ u'_i$. When $t = \texttt{bool}$, by definition sim we obtain, for each i, $u_i = u'_i$. Thus $\mathsf{sim}\ t\ (\bigsqcup_i u_i)\ (\bigsqcup_i u'_i)$.

When $t = t_1 \to t_2$, consider any P, S, d, d' with $P = \mathsf{privs}(S)$ and $\mathsf{sim}\ t_1\ d\ d'$. We must show $\mathsf{sim}\ t_2\ ((\bigsqcup_i u_i)Pd)\ ((\bigsqcup_i u'_i)Sd')$, *i.e.*, by definition of lubs we must show, $\mathsf{sim}\ t_2\ \bigsqcup_i(u_iPd)\ \bigsqcup_i(u'_iSd')$. By assumption, for every i, $\mathsf{sim}\ (t_1 \to t_2)\ u_i\ u'_i$, hence, $\mathsf{sim}\ t_2\ (u_iPd)\ (u'_iSd')$ holds for each i. Therefore, by induction for t_2, we obtain $\mathsf{sim}\ t_2\ \bigsqcup_i(u_iPd)\ \bigsqcup_i(u'_iSd')$. ■

Lemma 7.5 For any stack $(\langle n, P'\rangle :: S)$, for any standard expression e, and any D, t, h, h', let $u = [\![D \vdash e : t]\!]nPh$ where $P = \mathsf{privs}(\langle n, P'\rangle :: S)$, and let $u' = (\![D \vdash e : t]\!)(\langle n, P'\rangle :: S)h'$. Then $\mathsf{sim}\ D\ h\ h' \Rightarrow \mathsf{sim}\ t\ u\ u'$.

The Consistency Theorem follows from the lemma because $\mathsf{sim}\ \varnothing\ \{\}\ \{\}$ and since $\mathsf{sim}\ \texttt{bool}\ u\ u'$ implies $u = u'$.

Proof: of Lemma. Go by induction on e.

- Cases `true` and x: Immediate from semantic definitions.
- Case $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$: Directly by induction.
- Case $\texttt{fun}\ x.\ e$: Let $u = [\![D \vdash \texttt{fun}\ x.\ e : t_1 \to t_2]\!]nPh$ and let

$$u' = (\![D \vdash \texttt{fun}\ x.\ e : t_1 \to t_2]\!)Sh'$$

Then $u = \lambda P'.\ \lambda d.\ [\![D, x : t_1 \vdash e : t_2]\!]nP'[h \mid x \mapsto d]$
$u' = \lambda S'.\ \lambda d'.\ (\![D, x : t_1 \vdash e : t_2]\!)S'[h' \mid x \mapsto d']$
To show $\mathsf{sim}\ (t_1 \to t_2)\ u\ u'$, need to show that for any S'', d'', d''', such that $\mathsf{sim}\ t_1\ d''\ d'''$, it is the

case that $\mathsf{sim}\ t_2\ (u\ (\mathsf{privs}\ S'')\ d'')\ (u'\ S''\ d''')$. By standardness, e is $\mathtt{signs}\ n'\ e'$ for some n', e'. Thus we can proceed as follows, using $e \equiv \mathtt{signs}\ n'\ e'$ and semantics of $\mathtt{signs}$.

$$\begin{array}{rcl} u\ (\mathsf{privs}\ S'')\ d'' & = & [\![D, x : t_1 \vdash e : t_2]\!] n\ (\mathsf{privs}\ S'')\ [h \mid x \mapsto d''] \\ & = & [\![D, x : t_1 \vdash e' : t_2]\!] n'\ (\mathsf{privs}(S'') \cap \mathscr{A}(n'))\ [h \mid x \mapsto d''] \\ u'\ S''\ d''' & = & ([D, x : t_1 \vdash e : t_2])\ S''\ [h' \mid x \mapsto d'''] \\ & = & ([D, x : t_1 \vdash e' : t_2])(\langle n', \varnothing\rangle :: S'')\ [h' \mid x \mapsto d'''] \end{array}$$

Note that by definition sim and by assumption $\mathsf{sim}\ t_1\ d''\ d'''$, we have, $\mathsf{sim}\ (D, x : t_1)\ [h \mid x \mapsto d'']\ [h' \mid x \mapsto d''']$. Furthermore, by Fact 7.1, $\mathsf{privs}(S'') \cap \mathscr{A}(n') = \mathsf{privs}(\langle n', \varnothing\rangle :: S'')$. Therefore, by induction for e', we obtain, $\mathsf{sim}\ t_2\ (u\ (\mathsf{privs}\ S'')\ d'')\ (u'\ S''\ d''')$. This is where we need Definition 2.1.

- Case $e_1\, e_2$: $\begin{array}{rcl} [\![D \vdash e_1\ e_2 : t_2]\!] nPh & = & \textbf{let}\ f\ =\ [\![D \vdash e_1 : t_1 \to t_2]\!] nPh\ \textbf{in} \\ & & \textbf{let}\ d = [\![D \vdash e_2 : t_1]\!] nPh\ \textbf{in}\ fPd \\ ([D \vdash e_1\ e_2 : t_2])Sh' & = & \textbf{let}\ f' =\ ([D \vdash e_1 : t_1 \to t_2])Sh'\ \textbf{in} \\ & & \textbf{let}\ d' = ([D \vdash e_2 : t_1])Sh\ \textbf{in}\ f'Sd' \end{array}$

  Need to show $\mathsf{sim}\ t_2\ (fPd)\ (f'Sd')$. Since $\mathsf{sim}\ D\ h\ h'$ and $P = \mathsf{privs}(S)$, therefore, by induction for e_1, we have $\mathsf{sim}\ (t_1 \to t_2)\ f\ f'$. Similarly, by induction for e_2, we have $\mathsf{sim}\ t_1\ d\ d'$. Hence the result follows by definition sim since $P = \mathsf{privs}(S)$. This case of the proof shows the necessity of defining the relation sim.

- Case $\mathtt{letrec}\ f(x) = e_1\ \mathtt{in}\ e_2$:

$$\begin{array}{l} [\![D \vdash \mathtt{letrec}\ f(x) = e_1\ \mathtt{in}\ e_2 : t]\!] nPh \\ \quad = \ \textbf{let}\ G(g)\ =\ \lambda P'.\, \lambda d.\, [\![D, f : t_1 \to t_2, x : t_1 \vdash e_1 : t_2]\!] nP'[h \mid f \mapsto g, x \mapsto d]\ \textbf{in} \\ \quad\quad [\![D, f : t_1 \to t_2 \vdash e_2 : t]\!] nP[h \mid f \mapsto \mathit{fix}\ G] \\ ([D \vdash \mathtt{letrec}\ f(x) = e_1\ \mathtt{in}\ e_2 : t])Sh \\ \quad = \ \textbf{let}\ G'(g')\ =\ \lambda S'.\, \lambda d'.\, ([D, f : t_1 \to t_2, x : t_1 \vdash e_1 : t_2])S'[h' \mid f \mapsto g', x \mapsto d']\ \textbf{in} \\ \quad\quad ([D, f : t_1 \to t_2 \vdash e_2 : t])S[h' \mid f \mapsto \mathit{fix}\ G'] \end{array}$$

  To show the result, it suffices to show $\mathsf{sim}\ (t_1 \to t_2)\ (\mathit{fix}\ G)\ (\mathit{fix}\ G')$, because then we can use induction for e_2, noting that $\mathsf{sim}\ (D, f : t_1 \to t_2)\ [h \mid f \mapsto \mathit{fix}\ G]\ [h' \mid f \mapsto \mathit{fix}\ G']$, and that $P = \mathsf{privs}(S)$. Accordingly, we demonstrate the following claim:

$$\forall i.\mathsf{sim}\ (t_1 \to t_2)\ g_i\ g_i' \tag{7}$$

  Then from Lemma 7.4, we get $\mathsf{sim}\ (t_1 \to t_2)\ \bigsqcup_i g_i\ \bigsqcup_i g_i'$. This completes the proof. To show (7), we proceed by induction on i. We have:

$$\begin{array}{rcl} g_0 & = & \lambda P'.\, \lambda d.\, \bot \\ g_{i+1} & = & \lambda P'.\, \lambda d.\, [\![D, f : t_1 \to t_2, x : t_1 \vdash e_1 : t_2]\!] nP'[h \mid f \mapsto g_i, x \mapsto d] \\ & = & \{\text{because } e_1 \equiv \mathtt{signs}\ n'\ e_1' \text{ by standardness}\} \\ & & \lambda P'.\, \lambda d.\, [\![D, f : t_1 \to t_2, x : t_1 \vdash e_1 : t_2]\!] n'(P' \cap \mathscr{A}(n'))[h \mid f \mapsto g_i, x \mapsto d] \end{array}$$

$$\begin{array}{rcl} g_0' & = & \lambda S'.\, \lambda d'.\, \bot \\ g_{i+1}' & = & \lambda S'.\, \lambda d'.\, ([D, f : t_1 \to t_2, x : t_1 \vdash e_1 : t_2])S'[h' \mid f \mapsto g_i', x \mapsto d'] \\ & = & \{\text{because } e_1 \equiv \mathtt{signs}\ n'\ e_1'\} \\ & & \lambda S'.\, \lambda d'.\, ([D, f : t_1 \to t_2, x : t_1 \vdash e_1' : t_2])(\langle n', \varnothing\rangle :: S')[h' \mid f \mapsto g_i', x \mapsto d'] \end{array}$$

Clearly, $\mathsf{sim}\ (t_1 \to t_2)\ g_0\ g_0'$, by definition sim. To show $\mathsf{sim}\ (t_1 \to t_2)\ g_{i+1}\ g_{i+1}'$, assume $\mathsf{sim}\ (t_1 \to t_2)\ g_i\ g_i'$ (induction hypothesis), and that for any S' and $P' = \mathsf{privs}(S')$, $\mathsf{sim}\ t_1\ d\ d'$ holds. Then

$$\mathsf{sim}\ (D, f : t_1 \to t_2, x : t_1)\ [h \mid f \mapsto g_i, x \mapsto d]\ [h' \mid f \mapsto g_i', x \mapsto d']$$

by definition sim and since $\mathsf{sim}\ D\ h\ h'$. Now by Fact 7.1, $P' \cap \mathscr{A}(n') = \mathsf{privs}(\langle n', \varnothing\rangle :: S')$, so by the main induction hypothesis on e_1', $\mathsf{sim}\ t_2\ (g_{i+1} P' d)\ (g_{i+1}' S' d')$ holds.

- Case `signs` $n\ e$: We have: $[\![D \vdash \texttt{signs}\ n'\ e : t]\!] n P h = [\![D \vdash e : t]\!] n' (P \cap \mathscr{A}(n'))h$ and $([D \vdash \texttt{signs}\ n'\ e : t])Sh' = ([D \vdash e : t])(\langle n', \varnothing\rangle :: S)h'$ so the result holds by induction on e provided $P' \cap \mathscr{A}(n') = \mathsf{privs}(\langle n', \varnothing\rangle :: S)$. But this equality holds by Fact 7.1.
- Case `dopriv` p `in` e: The result holds by induction for e, provided that $P \sqcup_n \{p\} = \mathsf{privs}(\langle n, P' \cup \{p\}\rangle :: S)$. This holds because for any p'

	$p' \in P \sqcup_n \{p\}$	
$\Leftrightarrow$	$p' \in P \vee (p' \in \mathscr{A}(n) \wedge p' = p)$	by def $\sqcup_n$
$\Leftrightarrow$	$\mathsf{chk}(p', \langle n, P'\rangle :: S) \vee (p' \in \mathscr{A}(n) \wedge p' = p)$	assumption, def privs
$\Leftrightarrow$	$(p' \in \mathscr{A}(n) \wedge (p' \in P' \vee \mathsf{chk}(p', S)) \vee (p' \in \mathscr{A}(n) \wedge p' = p)$	def chk
$\Leftrightarrow$	$p' \in \mathscr{A}(n) \wedge (p' \in P' \cup \{p\} \vee \mathsf{chk}(p', S))$	logic and sets
$\Leftrightarrow$	$p' \in \mathsf{privs}(\langle n, P' \cup \{p\}\rangle :: S)$	defs chk and privs

- Case `check` p `for` e: Both semantics are conditional; the condition in one case is $p \in P'$ and in the other case $\mathsf{chk}(p, S)$, and these are equivalent conditions by assumption $P' = \mathsf{privs}(S)$ for the Lemma. In case the condition is true, the result holds by induction, which applies because for both semantics the security arguments for e are unchanged. If the condition is false, the result holds because both semantics are $\star$ and $\mathsf{sim}\ t\ \star\ \star$.
- Case `test` p `then` e_1 `else` e_2: Similar to the case for `check`.

8 Conclusion

Our work was motivated by the hope, inspired by discussions with Dave Schmidt, for more principled semantics of static analyses presented in the form of type and effect systems. Our work serves to demonstrate two attractive features of denotational semantics, which a decade ago seemed largely eclipsed by operational semantics. The first is proof of program equalities via equational reasoning on denotations. The second is logical relations, defined by structural recursion on types. We are glad for the opportunity to demonstrate the utility of denotational semantics while celebrating the contributions of Dave Schmidt, so adept a practitioner of all forms of semantic modeling.

References

[1] M. Abadi, M. Burrows, B. W. Lampson & G. D. Plotkin (1993): *A Calculus for Access Control in Distributed Systems*. *ACM Transactions on Programming Languages and Systems* 15(4), pp. 706–734, doi:10.1145/155183.155225.

[2] M. S. Ager, O. Danvy & J. Midtgaard (2005): *A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects*. *Theoretical Computer Science* 342, pp. 4–28, doi:10.1016/j.tcs.2005.06.008.

[3] A. Banerjee & D. A. Naumann (2001): *A Simple Semantics and Static Analysis for Java Security*. Technical Report CS Report 2001-1, Stevens Institute of Technology. Available at `http://www.cs.stevens-tech.edu/~naumann/tr2001.ps`.

[4] A. Banerjee & D. A. Naumann (2002): *Representation Independence, Confinement and Access Control*. In: *ACM Symposium on Principles of Programming Languages*, pp. 166–177, doi:10.1145/503272.503289.

[5] A. Banerjee & D. A. Naumann (2005): *Stack-based Access Control for Secure Information Flow*. *Journal of Functional Programming* 15(2), pp. 131–177, doi:10.1017/S0956796804005453.

[6] F. Besson, T. Blanc, C. Fournet & A. D. Gordon (2004): *From Stack Inspection to Access Control: A Security Analysis for Libraries*. In: *Computer Security Foundations Workshop (CSFW)*, pp. 61–75, doi:10.1109/CSFW.2004.11.

[7] F. Besson, T. P. Jensen & D. Le Métayer (2001): *Model Checking Security Properties of Control Flow Graphs*. *Journal of Computer Security* 9(3), pp. 217–250.

[8] F. Besson, T. de Grenier de Latour & T. P. Jensen (2005): *Interfaces for Stack Inspection*. *Journal of Functional Programming* 15(2), pp. 179–217, doi:10.1017/S0956796804005465.

[9] P. Centonze, R. J. Flynn & M. Pistoia (2007): *Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-Control Policies*. In: *Computer Security Applications Conference (ACSAC)*, pp. 292–303, doi:10.1109/ACSAC.2007.14.

[10] J. Clements & M. Felleisen (2004): *A Tail-recursive Machine with Stack Inspection*. *ACM Transactions on Programming Languages and Systems* 26(6), pp. 1029–1052, doi:10.1145/1034774.1034778.

[11] C. Fournet & A. D. Gordon (2003): *Stack inspection: Theory and Variants*. *ACM Transactions on Programming Languages and Systems* 25(3), pp. 360–399, doi:10.1145/641909.641912.

[12] L. Gong (1999): *Inside Java 2 Platform Security*. Addison-Wesley.

[13] B. A. LaMacchia, S. Lange, M. Lyons, R. Martin & K. T. Price (2002): *.NET Framework Security*. Addison-Wesley.

[14] M. Pistoia, A. Banerjee & D. A. Naumann (2007): *Beyond Stack Inspection: A Unified Access-Control and Information-Flow Security Model*. In: *28th IEEE Symposium on Security and Privacy*, pp. 149–163, doi:10.1109/SP.2007.10.

[15] F. Pottier, C. Skalka & S. Smith (2005): *A Systematic Approach to Static Access Control*. *ACM Transactions on Programming Languages and Systems* 27(2), pp. 344–382, doi:10.1145/1057387.1057392.

[16] C. Skalka (2002): *Types for Programming Language-Based Security*. Ph.D. thesis, The Johns Hopkins University.

[17] C. Skalka & S. Smith (2000): *Static Enforcement of Security with Types*. In: *International Conference on Functional Programming*, pp. 34–45, doi:10.1145/351240.351244.

[18] C. Skalka, S. Smith & D. Van Horn (2008): *Types and Trace Effects of Higher Order Programs*. *Journal of Functional Programming* 18(2), pp. 179–249, doi:10.1017/S0956796807006466.

[19] D. Wallach, A. Appel & E. Felten (2000): *SAFKASI: A Security Mechanism for Language-based Systems*. *ACM Transactions on Software Engineering and Methodology* 9(4), pp. 341–378, doi:10.1145/363516.363520.