Dynamic Security Labels and Noninterference

Lantian Zheng Andrew C. Myers
Computer Science Department
Cornell University
{zlt,andru}@cs.cornell.edu

Abstract

This paper explores information flow control in systems in which the security classes of data can vary dynamically. Information flow policies provide the means to express strong security requirements for data confidentiality and integrity. Recent work on security-typed programming languages has shown that information flow can be analyzed statically, ensuring that programs will respect the restrictions placed on data. However, real computing systems have security policies that vary dynamically and that cannot be determined at the time of program analysis. For example, a file has associated access permissions that cannot be known with certainty until it is opened. Although one security-typed programming language has included support for dynamic security labels, there has been no examination of whether such a mechanism can securely control information flow. In this paper, we present an expressive language-based mechanism for securely manipulating dynamic security labels. The mechanism is presented both in the context of a Java-like programming language and, more formally, in a core language based on the typed lambda calculus. This core language is expressive enough to encode previous dynamic label mechanisms; as importantly, any well-typed program is provably secure because it satisfies noninterference.

1 Introduction

Information flow control protects information security by constraining how information is transmitted among objects and users of various security classes. These security classes are expressed as *labels* associated with the information or its containers. *Dynamic labels*, labels that can be manipulated and checked at run time, are vital for modeling real systems in which security policies may be changed dynamically. For example, it is important to be able to change security settings on files and database records, and these changes should affect how the information from these sources can be used.

However, manipulating labels dynamically makes it difficult to enforce a strong notion of information security such as *noninterference* [8] for several reasons. First, downgrading the label of an object may convert sensitive data to public data, directly violating noninterference. Second, label changes can be used to convey information covertly; some restriction has to be imposed to prevent such covert channels [28, 21]. Third, the usual way to control information flow in the presence of dynamic labels is mandatory access control (MAC), which generally cannot prevent *implicit flows* arising from the control flow paths not taken at run time [4, 12].

Static information flow control techniques, such as those developed by Denning and Denning [5], are able to analyze all control flow paths and prevent illegal implicit flows. Moreover, static information flow analysis incurs little run-time overhead. Recently, static information flow analyses have been formalized in *security type systems* (e.g., [26, 10, 32, 18, 3, 20]) that can provably enforce noninterference. Nevertheless, in most security-typed languages, security labels are purely static type-level information that cannot be accessed or tested at run time.

JFlow [14] and its successor, Jif [16] are the only security-typed languages supporting dynamic labels. However, although the Jif type system is designed to control the new information channels that dynamic labels create, it has not been proved to enforce secure information flow. Further, the dynamic label mechanism in Jif has limitations that impair expressiveness and efficiency.

In this paper, we propose a new, expressive language-based mechanism for securely manipulating dynamic security labels. We show that the mechanism is useful in practice by incorporating it into Jif and demonstrating that the resulting language, Jif-DX, is more expressive than Jif. Further, we study the soundness of this mechanism by formalizing it in a core language based on the typed lambda calculus and proving that any well-typed program of the core language is secure because it satisfies noninterference. This is the first noninterference proof for a security-typed language in which general security labels can be manipulated and tested dynamically.

While downgrading security classes [24, 15] is an important capability, it is useful to treat it as a separate mechanism so that labels can be manipulated dynamically while preserving noninterference. However, the Jif language does support a downgrading mechanism *without* resorting to dynamic labels.

The remainder of this paper is organized as follows. Section 2 presents some background on lattice label models and dynamic labeling. Section 3 introduces the static analysis framework for dynamic labels and the Jif-DX language. Section 4 formalizes the static analysis of dynamic labels as the type system of a core language λ_{DSec} and proves the noninterference result. Section 5 covers related work, and Section 6 concludes.

2 Background

2.1 Security classes

We assume that security requirements for confidentiality or integrity are defined by associating *security classes* with users and with the resources that programs access. These security classes form a lattice $\mathcal{L}$. We write $k \sqsubseteq k'$ to indicate that security class k' is at least as restrictive as another security class k. In this case it is safe to move information from security class k to k', because restrictions on the use of the data are preserved. To control data derived from sources with classes k and k', the least restrictive security class that is at least as restrictive as both k and k' is assigned. This is the least upper bound, or join, written $k \sqcup k'$.

2.2 Labels

Type systems for confidentiality or integrity are concerned with tracking information flows in programs. Types are extended with security *labels* that denote security classes. A label ℓ appearing in a program may be simply a constant security class k, or a more complex expression that denotes a security class. The notation $\ell_1 \sqsubseteq \ell_2$ means that ℓ_2 denotes a security class that is at least as restrictive as that denoted by ℓ_1.

Because a given security class may be denoted by different labels, the relation $\sqsubseteq$ generates a lattice of *equivalence classes* of labels with $\sqcup$ as the *join* (least upper bound) operator. Two labels ℓ_1 and ℓ_2 are equivalent, written $\ell_1 \approx \ell_2$, if $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_1$. The join of two labels, $\ell_1 \sqcup \ell_2$, denotes the security class that is the join of the security classes that ℓ_1 and ℓ_2 denote. For example, if `x` has label ℓ_x and `y` has label ℓ_y, then the sum `x+y` is given the label $\ell_x \sqcup \ell_y$.

2.3 Security type systems for information flow

Security type systems can be used to enforce security information flows statically. Information flows in programs may be explicit flows such as assignments, or implicit flows [5] arising from the control flow of the program. Consider an assignment statement `x=y`, which contains an information flow from `y` to `x`. Then the typing rule for the assignment statement requires that $\ell_y \sqsubseteq \ell_x$, which means the security level of `y` is lower than the security level of `x`, guaranteeing the information flow from `y` to `x` is secure.

One advantage of static analysis is more precise control of implicit flows. Consider a simple conditional:

$$\texttt{if b then x} = \texttt{true else x} = \texttt{false}$$

After running this expression, the value of x is equal to b, although there is no direct assignment from b to x. A standard technique to prevent illegal implicit flows is to introduce a *program-counter label* [4], written *pc*, which indicates the security level of the information that can be learned by knowing the control flow path taken thus far. The type system ensures that any effect of expression e has a label at least as restrictive as its *pc*. In other words, expression e cannot generate any effects observable to users who should not know the current program counter.

2.4 Dynamic labels in Jif

Jif [16] (previously known as JFlow [14]) is the only existing security-typed language that supports dynamic labels. Jif extends the Java language [9] with security labels that are based on the *decentralized label model* [15]. These labels may explicitly mention principals. For example, a value with type `int{Alice:Bob}` is an integer owned by principal `Alice` and readable by `Alice` and `Bob`. Jif aims to provide a usable programming model, in which the dynamic label mechanism plays an important role.

In Jif, security labels can be used as first-class values, so labels are not purely static type annotations. In addition, variables of type `label` (*label variables*) may be used as a label for other values. Label variables provide a straightforward way to represent dynamic labels. For example, suppose x is a label variable. Then `*x` refers to the label contained in x, and `{*x}` is a legitimate label in Jif if x is declared `final` so that it cannot be assigned after initialization, and the meanings of labels do not change as variables are assigned. Dynamic labels are treated as unknown but *fixed* labels by the compiler, so they can be propagated in static checking. For example, given any two labels ℓ_1 and ℓ_2 such that $\ell_1 \sqsubseteq \ell_2$, it is the case that $\ell_1 \sqcup \texttt{\{*x\}} \sqsubseteq \ell_2 \sqcup \texttt{\{*x\}}$.

Since dynamic labels are generally unknown at compile time, it may be impossible to decide statically whether $\ell_1 \sqsubseteq \ell_2$ holds. In this case, the condition $\ell_1 \sqsubseteq \ell_2$ can only be enforced by examining labels at run time. For example, suppose a program tries to send an integer through a network channel that is created at run time and has a dynamic label. The operation is safe only if the label of the channel is at least as high as the label of the integer; this condition can only be tested at run time.

Jif provides the `switch label` statement for run-time label tests. The following code shows how to implement the above example using the `switch label` statement:

(A)
```
final label{} x;
Channel{*x} c;
int{Alice:} y;
switch label(y) {
    case (int{*x} z) c.send(z);
    else throw new UnsafeTransfer();
}
```

The label of channel c is a dynamic label `{*x}`. The label of x is the bottom label `{}`, which means the information about x is public. The `switch label` statement executes the first of the cases whose associated label is at least as restrictive as that of y. The value of y is assigned to the corresponding variable (for example, z). Thus the `send` operation will be executed only if $\texttt{\{Alice:\}} \sqsubseteq \texttt{\{*x\}}$, guaranteeing that c is a secure channel for sending y.

Like labels, principals may also be used as first-class values at run time. The statement `actsFor(p1, p2)`S executes the statement S if the principal p1 can act for the principal p2. This acts-for relationship between p1 and p2 is equivalent to $\texttt{\{p2:\}} \sqsubseteq \texttt{\{p1:\}}$. Thus the `actsFor` statement essentially implements a run-time label examination.

3 Static analysis of dynamic labels

This section presents a general framework for static checking of dynamic labels. We propose the language Jif-DX, which extends Jif with a more expressive dynamic label mechanism based on this framework.

3.1 Static checking framework for dynamic labels

Static checking of dynamic labels must rely on the information about dynamic labels available at compile time. The insight behind the new static checking framework is to represent this information as *label constraints* of the form $\ell_1 \sqsubseteq \ell_2$. For example, the constraint `{Alice:}` $\sqsubseteq$ `{*x}` indicates that the label contained in `x` is at least as restrictive as `{Alice:}`. Then it is safe to assign a value of label `{Alice:}` to a variable of label `{*x}`, even though the exact value of `x` is unknown.

For a security-typed language, static information flow checking is an aspect of type checking, which ensures that a well-typed expression does not generate illegal information flows. In general, if type-checking an expression e involves dynamic labels, the compiler can reason more accurately about information flow by exploiting the set of label constraints known to be satisfied when e is executed. Thus, tracking and using the label constraints of each expression is the key to improving static checking of dynamic labels. Essentially, a label constraint is a kind of type constraint, which has been used in bounded polymorphic types, type inference and dependent type systems [23, 30].

We can classify label constraints into three categories: dynamic constraints, static constraints and implicit constraints. This classification helps identifying various label constraints systematically and provides hints for new dynamic label constructs.

- **Dynamic label constraints**
  Dynamic label constraints are constraints enforced by testing labels at run time. For example, consider the `switch label` statement: `switch label(`e`) {`$\ldots$`case (T{`ℓ`} y):` S $\ldots$`}`. If S is executed, then $\ell_e \sqsubseteq \ell$ (ℓ_e represents the label of e) must be satisfied, and thus, the constraint could be used in type-checking S. However, Jif does not make use of this constraint when statically checking S.

- **Static label constraints**
  Static label constraints are constraints enforced statically by the compiler. For example, in Jif, an `actsFor` constraint "`p1 actsFor p2`" may be specified in a method signature to prevent the method from being called unless the compiler can determine that principal `p1` acts for `p2` at the call site [14]. This `actsFor` constraint is similar to a static label constraint `{p2:}` $\sqsubseteq$ `{p1:}`, though it has some separate utility in Jif.

  One advantage of static constraints is that no run-time cost is incurred because they are enforced statically. Furthermore, not all the static constraints can be enforced dynamically because some labels such as *class label parameters* [14] have no run-time representations.

- **Implicit label constraints**
  Implicit label constraints are not explicitly specified in programs, but can be inferred from programs. For example, consider the statement "`final label lb =` ℓ". It is clear that the constraint `{*lb}` $\approx$ ℓ holds after the statement is executed. Implicit label constraints can be used to type-check call expressions. In the following code, the type of expression `o.m({Alice:},10)` is `int{Alice:}` because of an implicit constraint `{*lb}` $\approx$ `{Alice:}` that arises from argument passing.

  (B)
  ```
  interface I { int{*lb} m(label{} lb, int{*lb} x); }
  I{} o; ...
  int{Alice:} y = o.m({Alice:}, 10);
  ```

  To enable the compiler to generate an implicit label constraint for every actual label argument, Jif imposes a syntactic restriction on the method argument of type `label`: the actual label argument in

a call expression must be an expression that can be converted to a label without evaluation. This restriction does not substantively affect expressiveness. For example, given an arbitrary expression e of type `label`, the expression `o.m(`e`,10)` can be rewritten as "`final label t =` e`; o.m(t,10)`", where `t` can be converted to the label `{*t}`.

3.2 The Jif-DX language

The original Jif dynamic label mechanism appears to be sound but has several limitations. First, label checking of the clauses of a `switch label` statement does not fully exploit the label constraint enforced by the run-time check. Second, Jif supports only one kind of static label constraint: `actsFor` constraints, which give information about principals but are not as powerful as general label constraints. Third, in Jif only label variables can be used as dynamic labels, but in practice other expressions may be useful in dynamic labels.

These limitations of Jif make it difficult or awkward to write some applications that need to manipulate dynamic labels. Therefore, we propose the Jif-DX language, which extends Jif with a better dynamic label mechanism, including the label-test statement, method and field label constraints, and more general label expressions.

3.2.1 The label-test statement

Jif-DX provides the label-test statement, which is a more flexible way to implement run-time label checks than the `switch label` statement. The syntax of the label-test statement resembles a normal `if` statement, except that the conditional expression must be a label constraint syntactically: "`if (`ℓ_1`<=`ℓ_2`)` S_1 `else` S_2". Intuitively, S_1 is executed if $\ell_1 \sqsubseteq \ell_2$ is true at run time; otherwise, S_2 is executed. Because $\ell_1 \sqsubseteq \ell_2$ must hold if S_1 is executed, this constraint can be assumed to hold when checking S_1 statically.

Both the `switch label` statement and the `actsFor` statement in Jif can be encoded with the label-test statement. For example, the statement "`actsFor(p1, p2)` S" is equivalent to "`if ({p2:} <= {p1:})` S".

3.2.2 Method label constraints

Jif-DX allows general label constraints to be specified in method signatures, whereas Jif only provides `actsFor` constraints. The following example shows a use of a label constraint on a method:

(C)
```
class Key[principal p] {
    int{} encrypt(label{} lb, int{*lb} x) where {*lb} <= {p:}   { ...  }
}
```

The class `Key[principal p]` represents a key belonging to principal `p`. The `encrypt` method takes in a label `lb` and an integer `x` labeled with `{*lb}`, and attempts to encrypt `x` with the key of principal `p` and return the encrypted result as a public integer. This method should only encrypt the data owned by principal `p`, because the result can be decrypted by `p`. This requirement is captured by the method label constraint `{*lb}` $\sqsubseteq$ `{p:}`. The compiler ensures that the constraint is satisfied wherever this method is called.

Another way to write this code would be to insert a run-time check in the method body and make the method throw an exception if `{*lb}` $\sqsubseteq$ `{p:}` is not satisfied at run time. This code would incur some unnecessary run-time label checks, and the caller would have to handle the exception somehow. Indeed, one advantage of the method label constraint is its ability to exploit information available at the caller side to reduce the number of run-time checks. For example, in the following Jif-DX code the compiler can determine that the method constraint is satisfied without a run-time check:

(D)
```
Key[Alice]{} k;
int{Alice:Bob} x;
k.encrypt({Alice:Bob}, x);
```

3.2.3 Field label constraints

In Jif-DX, label constraints can also be specified on class fields of type `label`. The compiler ensures that the field label constraints of a class are satisfied whenever a new instance of the class is created. All fields appearing in a label constraint must be final, so field label constraints that are satisfied when an object is created will hold for the lifetime of the object.

Like method label constraints, field label constraints can be used to reduce the number of run-time label checks. For example, sending an integer through a *multilevel communication channel* [6] with label ℓ requires sending the exact label of the integer through the channel. The natural way to implement it is to wrap the integer and its label in an object of the `Labeled` class and send the object through the channel.

(E)
```
class Labeled {
   public final label{ℓ} lb;
   public int{*lb} content;
   public Labeled(label{ℓ} x, int{*x} y) { lb = x;  content = y; }
}
```

The label of field `lb` is ℓ, ensuring that `lb` itself can be sent through the channel. But the label of field `content` is dynamic, and the constraint `{*lb}` $\sqsubseteq$ ℓ needs to hold for field `content` to be sent safely through the channel. This constraint can be enforced by a run-time label check, but it can also be enforced statically by specifying a field label constraint `{*lb}` $\sqsubseteq$ ℓ, as in the `UBLabeled` ("`UB`" stands for upper bound) class. Sending a `UBLabeled` object through a channel with label ℓ is always safe.

(F)
```
class UBLabeled {
   public final label{ℓ} lb where {*lb} <= ℓ;
   public int{*lb} content;
   public UBLabeled(label{ℓ} x, int{*x} y) where {*x} <= ℓ {
       lb = x;  content = y;
   }
}
```

3.2.4 Path-expression labels

Consider the `Labeled` class again, and suppose `o` is a `Labeled` object. Then what is the type of `o.content`? According to the `Labeled` class, the precise type would be `int{*o.lb}`, which cannot be expressed in Jif because Jif does not allow *path expressions* such as `o.lb` to appear in labels.

In Jif-DX, a path expression with the type `label` can be used in label expressions as long as all the identifiers in the path expression are final, ensuring that the path expression always has the same value. For example, if `o` is a final variable, then `{*o.lb}` is a legitimate label, and the following code can be used to access `o.content` while preserving its precise type.

(G)
```
int{*o.lb} y = o.content;
```

If `o` were not a final variable, then `o.content` would not be well-typed in Jif-DX. But there is an easy workaround: assign `o` to a final variable `fo` and access the `content` field by `fo.content`, which has a well-formed type `int{*fo.lb}`.

3.2.5 Example: bounded dynamic labeling

In this section, we show how to use the new dynamic label constructs in Jif-DX to implement a MAC mechanism, which would be much harder and unintuitive to implement in Jif. The MAC mechanism in the MITRE CMW system [28] associates two labels with each object: a *floating label* and a fixed *mandatory*

label. The floating label is updated accordingly when the content of the object is updated, but is bounded by the fixed mandatory label in order to prevent the covert channel caused by label updates. The doubly labeled object can be represented by a `UBLabeled` (see code fragment F) object in Jif-DX, and the policy that the floating label be bounded by the mandatory label is represented by the field constraint `{*lb}` $\sqsubseteq \ell$, where `{*lb}` is the floating label, and ℓ is the mandatory label.

The following code shows how to update the label and access the content of a `UBLabeled` object. Simple as it is, this example demonstrates several subtle issues related to manipulating dynamic labels.

```
(H) UBLabeled o;
    final label{} x, y;
    int{*x} data;
    ...
(1) if ({*x} <= ℓ) o = new UBLabeled(x, data);
    final UBLabeled{} fo = o;
(2) if ({*fo.lb} <= {*y})
        if ({*y} <= ℓ) o = new UBLabeled(y, fo.content);

(3) int{ℓ} output = fo.content;
    int{Alice:} output2;
(4) if ({*fo.lb} <= {Alice:}) output2 = fo.content;
```

The first label-test statement (1) attempts to update the content of `o`, and the constraint `{*x}` <= ℓ guarantees the label of the new value is bounded by the mandatory label ℓ. The constructor call `new UBLabeled(x, data)` is well-typed because of the constraint `{*x}` $\sqsubseteq \ell$ enforced by the label test.

The second label-test statement (2) attempts to just update the label field of `o` to `y`. The first test `{*fo.lb}` <= `{*y}` is necessary for `new UBLabeled(y, fo.content)` to be well-typed, because the type of `fo.content` (`int{*fo.lb}`) must be a subtype of `int{*y}`. Essentially, the constraint prevents downgrading the label of the object content. Furthermore, this example shows that the immutability requirement for label fields is not a fundamental limitation because adding a level of indirection makes it possible to update `o.lb` even though the field `lb` is final.

The last two statements (3,4) attempt to access `o.content`. The assignment to `output` is well-typed because of the field label constraint `{*fo.lb}` $\sqsubseteq \ell$. The assignment to `output2` might appear secure because a label test is used to ensure the label of `output2` is at least as restrictive as the label of `fo.content`. However, there is an implicit flow from `fo.lb` to `output2` in the label-test statement. The implicit flow is legal only if $\ell \sqsubseteq$ `{Alice:}`, which prevents a possible covert channel caused by dynamic labeling.

4 Type system and noninterference

This section formalizes the powerful dynamic label mechanism of Jif-DX and proves its soundness in term of enforcing noninterference, which means that high-security inputs to a program cannot affect low-security outputs.

The vehicle for this formal analysis is a core language λ_{DSec} focused on modeling the dynamic label constructs in Jif-DX. Distilling Jif-DX to a simple core language has the advantage that the semantics of the dynamic label mechanism can be described clearly and formally. Many features of Jif-DX are intentionally omitted from λ_{DSec}, including objects, class inheritance, exceptions, and downgrading; however, these features are largely orthogonal to the dynamic label mechanism, and their impact on information flow has been studied in other work [3, 22, 31].

$$
\begin{array}{rrcl}
\text{Base Labels} & k & \in & \mathcal{L} \\
\text{Variables} & x, y, f & \in & \mathcal{V} \\
\text{Locations} & m & \in & \mathcal{M} \\
\text{Labels} & \ell, \mathit{pc} & ::= & k \mid x \mid \ell_1 \sqcup \ell_2 \\
\text{Constraints} & C & ::= & \ell_1 \sqsubseteq \ell_2 \,,\, C \mid \epsilon \\
\text{Base Types} & \beta & ::= & \texttt{int} \mid \texttt{label} \mid \texttt{unit} \mid (x{:}\tau_1)[C] * \tau_2 \mid \tau\ \texttt{ref} \mid (x{:}\tau_1) \xrightarrow{C\,;\,\mathit{pc}} \tau_2 \\
\text{Security Types} & \tau & ::= & \beta_\ell \\
\text{Values} & v & ::= & x \mid n \mid m^\tau \mid \lambda(x{:}\tau)[C\,;\mathit{pc}].\, e \mid () \mid k \mid (x{=}v_1[C],\, v_2{:}\tau) \\
\text{Expressions} & e & ::= & v \mid \ell_1 \sqcup \ell_2 \mid e_1\, e_2 \mid\, !e \mid e_1 := e_2 \mid \texttt{ref}^\tau e \mid \texttt{if}\ \ell_1 \sqsubseteq \ell_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & & \mid & \texttt{let}\ (x, y){=}v\ \texttt{in}\ e
\end{array}
$$

Figure 1: Syntax of λ_{DSec}

4.1 The λ_{DSec} language

The λ_{DSec} language is a security-typed lambda calculus that supports first-class dynamic labels. In λ_{DSec}, labels are terms so that they can be manipulated and checked at run time. Furthermore, label terms can be used as type annotations that are analyzed statically. Syntactic restrictions are imposed on label terms to increase the practicality of type checking, which follows the approach used by Xi and Pfenning in $\texttt{ML}_0^\Pi(C)$ [30].

From the computational standpoint, λ_{DSec} is fairly expressive, because it supports both first-class functions and state (which together are sufficient to encode recursive functions).

4.1.1 Syntax

The syntax of λ_{DSec} is given in Figure 1. We use the name k to range over a lattice of label values $\mathcal{L}$ (more precisely, a join semi-lattice with bottom element $\bot$), x, y to range over variable names $\mathcal{V}$, and m to range over a space of memory addresses $\mathcal{M}$.

To make the lattice explicit, we write $\mathcal{L} \models k_1 \sqsubseteq k_2$ to mean that k_2 is at least as restrictive as k_1 in $\mathcal{L}$, and $\mathcal{L} \models k = k_1 \sqcup k_2$ to mean k is the join of k_1 and k_2 in $\mathcal{L}$. The least and greatest elements of $\mathcal{L}$ are $\bot$ and $\top$. We also assume $\mathcal{L}$ contains at least the points L and H where $H \not\sqsubseteq L$, but the noninterference result applies to an arbitrary lattice. The label L is assumed to describe what information is observable by *low-security users* who are to be prevented from seeing confidential information. Thus, *low-security* data has a label bounded above by L; *high-security* data has a label (such as H) not bounded by L.

In λ_{DSec}, a label can be either a label value k, a label variable x, or the join of two other labels $\ell_1 \sqcup \ell_2$. For example, L, x, and $L \sqcup x$ are all valid labels, and $L \sqcup x$ can be interpreted as a security policy that is as restrictive as both L and x. The security type $\tau = \beta_\ell$ is the base type β annotated with label ℓ. The base types include integers, unit, labels, functions, references and products.

The function type $(x : \tau_1) \xrightarrow{C\,;\,\mathit{pc}} \tau_2$ is a dependent type since τ_2, C and *pc* may mention x. To avoid recursion, x is not allowed to appear in τ_1. The component C is a set of *label constraints* with the form $\ell_1 \sqsubseteq \ell_2$, which must be satisfied when the function is invoked. The *pc* component is a lower bound on the memory effects of the function, and an upper bound on the *pc* label of the caller. Consequently, a function is not able to leak information about where it is called. Without the annotations C and *pc*, this kind of type is sometimes written as $\Pi x{:}\tau_1.\tau_2$ [13].

The product type $(x{:}\tau_1)[C] * \tau_2$ is also a dependent type in the sense that occurrences of x can appear in τ_2 and C. The component C is a set of label constraints that any value of the product type must satisfy. If τ_2 does not contain x, and C is empty, the type may be written as the more familiar $\tau_1 * \tau_2$. Without component C, this kind of type is sometimes written as $\Sigma x{:}\tau_1.\tau_2$ [13].

In λ_{DSec}, values include integers n, typed memory locations m^{τ}, functions $\lambda(x:\tau)[C\,;\mathit{pc}].\,e$, the unit value $()$, constant labels k, and pairs $(x = v_1[C],\ v_2:\tau)$. A function $\lambda(x:\tau)[C\,;\mathit{pc}].\,e$ has one argument x with type τ, and the components C and *pc* have the same meanings as those in function types. The empty constraint set C or the top *pc* can be omitted. A pair $(x = v_1[C],\ v_2:\tau)$ contains two values v_1 and v_2. The second element v_2 has type τ and may mention the first element v_1 by the name x. The component C is a set of label constraints that the first element of the pair must satisfy. For example, if C is $\{x \sqsubseteq L\}$, then $v_1 \sqsubseteq L$ must be true.

Expressions include values v, variables x, the join of two labels $\ell_1 \sqcup \ell_2$, applications $e_1\ e_2$, dereferences $!e$, assignments $e_1 := e_2$, references $\texttt{ref}^{\tau} e$, label-test expressions $\texttt{if}\ \ell_1 \sqsubseteq \ell_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, and product destructors $\texttt{let}\ (x,y) = v\ \texttt{in}\ e_2$.

The label-test expression $\texttt{if}\ \ell_1 \sqsubseteq \ell_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ is used to examine labels—at run time, if the value of ℓ_2 is a constant label at least as restrictive as the value of ℓ_1, then e_1 is evaluated, otherwise, e_2 is evaluated. Consequently, the constraint $\ell_1 \sqsubseteq \ell_2$ can be assumed when type-checking e_1.

The product destructor $\texttt{let}\ (x,y) = v\ \texttt{in}\ e$ unpacks the pair v, assigns the first element of v to x and the second to y, and then evaluates e.

4.1.2 Encoding Jif-DX constructs

The λ_{DSec} language is designed to model the dynamic label constructs of Jif-DX. Although λ_{DSec} is not object-oriented, first-class functions and products provide some ability to explore issues that arise in a class-based language (without inheritance).

The label-test statement in Jif-DX can be encoded directly by the label-test expression in λ_{DSec}. Methods in Jif-DX correspond to functions in λ_{DSec}, and both constructs allow constraints to be specified on the arguments of type `label`. Objects in Jif-DX correspond to product values in λ_{DSec}. Just as Jif-DX allows specifying label constraints on fields, λ_{DSec} allows constraints on product components. The λ_{DSec} language provides the product destructor to retrieve the components from a product value. This pattern-matching style of access not only retrieves the product components, but also preserves constraints.

The following λ_{DSec} expressions and types can be used to represent correspondingly labeled code fragments of Jif-DX in Section 3. Since class declarations in Jif-DX are essentially types, some Jif-DX code corresponds to types of λ_{DSec}. The λ_{DSec} type in (C) shows how to encode the signature of the method `encrypt`. The product types in (E) and (F) are used to encode the `Labeled` class and the `UBLabeled` class, respectively. The function term in (G) shows how to retrieve the components from a product value and use the components in some computation represented by e. The function in (H) encodes updating a `UBLabeled` object. It takes in three arguments: o is a reference of the product type encoding the `UBLabeled` class; y is a label; z is an integer labeled with y. The function wraps y and z in a product value and assigns the product value to o, updating the information contained in o and the corresponding label at the same time. A label-test expression is used to ensure that the product label constraint holds.

(C) $(x{:}\texttt{label}_{\bot}) \xrightarrow{x \sqsubseteq \{\mathsf{p}:\}} (y{:}\texttt{int}_x) \to \texttt{int}_{\bot}$

(E) $(x{:}\texttt{label}_{\ell}) * \texttt{int}_x$

(F) $(x{:}\texttt{label}_{\ell})[x \sqsubseteq \ell] * \texttt{int}_x$

(G) $\lambda o{:}((x{:}\texttt{label}_{\ell}) * \texttt{int}_x)_{\bot}.\,\texttt{let}\ (x,y) = o\ \texttt{in}\ e$

(H) $\lambda o{:}(((x{:}\texttt{label}_{\ell})[x \sqsubseteq \ell] * \texttt{int}_x)_{\ell}\ \texttt{ref})_{\bot}.\,\lambda y{:}\texttt{label}_{\ell}.\,\lambda(z{:}\texttt{int}_y)[\ell].$
$\texttt{if}\ y \sqsubseteq \ell\ \texttt{then}\ o := (x = y,\ z{:}\texttt{int}_x)\ \texttt{else}\ ()$

$$[E1] \quad \frac{\mathcal{L} \models k = k_1 \sqcup k_2}{\langle k_1 \sqcup k_2,\, M\rangle \longmapsto \langle k,\, M\rangle}$$

$$[E2] \quad \langle !m^\tau,\, M\rangle \longmapsto \langle M(m^\tau),\, M\rangle$$

$$[E3] \quad \frac{m = \mathit{newloc}(M)}{\langle \mathtt{ref}^\tau v,\, M\rangle \longmapsto \langle m^\tau,\, M[m^\tau \mapsto v]\rangle}$$

$$[E4] \quad \langle m^\tau := v,\, M\rangle \longmapsto \langle (),\, M[m^\tau \mapsto v]\rangle$$

$$[E5] \quad \langle (\lambda(x\!:\!\tau)[C\,;\mathit{pc}].\, e)\; v,\, M\rangle \longmapsto \langle e[v/x],\, M\rangle$$

$$[E6] \quad \frac{\mathcal{L} \models k_1 \sqsubseteq k_2}{\langle \mathtt{if}\; k_1 \sqsubseteq k_2\; \mathtt{then}\; e_1\; \mathtt{else}\; e_2,\, M\rangle \longmapsto \langle e_1,\, M\rangle}$$

$$[E7] \quad \frac{\mathcal{L} \models k_1 \not\sqsubseteq k_2}{\langle \mathtt{if}\; k_1 \sqsubseteq k_2\; \mathtt{then}\; e_1\; \mathtt{else}\; e_2,\, M\rangle \longmapsto \langle e_2,\, M\rangle}$$

$$[E8] \quad \langle \mathtt{let}\; (x,y) = (x = v_1[C],\, v_2\!:\!\tau)\; \mathtt{in}\; e,\, M\rangle \longmapsto \langle e[v_2/y][v_1/x],\, M\rangle$$

$$[E9] \quad \frac{\langle e,\, M\rangle \longmapsto \langle e',\, M'\rangle}{\langle E[e],\, M\rangle \longmapsto \langle E[e'],\, M'\rangle}$$

$$\begin{array}{rcl} E[\cdot] & ::= & [\cdot]\, e \mid v\, [\cdot] \mid [\cdot] := e \mid v := [\cdot] \mid\, !\,[\cdot] \mid \mathtt{ref}^\tau\, [\cdot] \mid [\cdot] \sqcup \ell_2 \mid k_1 \sqcup [\cdot] \\ & \mid & \mathtt{if}\; [\cdot] \sqsubseteq \ell_2\; \mathtt{then}\; e_1\; \mathtt{else}\; e_2 \mid \mathtt{if}\; k_1 \sqsubseteq [\cdot]\; \mathtt{then}\; e_1\; \mathtt{else}\; e_2 \end{array}$$

Figure 2: Small-step operational semantics of λ_{DSec}

4.1.3 Operational Semantics

The small-step operational semantics of λ_{DSec} is given in Figure 2. Let M represent a memory that is a finite map from typed locations to closed values, and let $\langle e,\, M\rangle$ be a machine configuration. Then a small evaluation step is a transition from $\langle e,\, M\rangle$ to another configuration $\langle e',\, M'\rangle$, written $\langle e,\, M\rangle \longmapsto \langle e',\, M'\rangle$.

It is necessary to restrict the form of $\langle e,\, M\rangle$ to avoid using undefined memory locations. Let $\mathit{loc}(e)$ represent the set of memory locations appearing in e. A memory M is well-formed if every address m appears at most once in $\mathit{dom}(M)$, and for any m^τ in $\mathit{dom}(M)$, $\mathit{loc}(M(m^\tau)) \subseteq \mathit{dom}(M)$. The configuration $\langle e,\, M\rangle$ is well-formed if M is well-formed, $\mathit{loc}(e) \subseteq \mathit{dom}(M)$, and e contains no free variables. By induction on the derivation of $\langle e,\, M\rangle \longmapsto \langle e',\, M'\rangle$, we can prove that if $\langle e,\, M\rangle$ is well-formed, then $\langle e',\, M'\rangle$ is also well-formed.

The notation $e[v/x]$ indicates capture-avoiding substitution of value v for variable x in expression e. The notation $M(m^\tau)$ denotes the value mapped to m^τ in M, and the notation $M[m^\tau \mapsto v]$ denotes the memory obtained by assigning v to m^τ in M.

The evaluation rules are standard. The allocator $\mathit{newloc}(M)$ in rule (E3) generates a fresh memory location m such that $m^\tau \notin \mathit{dom}(M)$ for all τ. In rule (E8), v_2 may mention x, so substituting v_2 for y in e is performed before substituting v_1 for x. The variable name in the product value matches x so that no variable substitution is needed when assigning v_1 and v_2 to x and y. In rule (E9), E represents an evaluation context, a term with a single "hole", into which a subterm can fit. Rule (E9) says that an evaluation step of a subterm counts as an evaluation step of the enclosing term. The syntax of E specifies the evaluation order of subterms.

$$[C1]\ \frac{\mathcal{L} \models k_1 \sqsubseteq k_2}{\vdash k_1 \sqsubseteq k_2} \qquad [C2]\ \frac{\ell_1 \sqsubseteq \ell_2 \in C}{C \vdash \ell_1 \sqsubseteq \ell_2} \qquad [C3]\ \vdash \ell \sqsubseteq \ell \sqcup \ell' \qquad [C4]\ \vdash \ell \sqsubseteq \ell$$

$$[C4]\ \frac{C \vdash \ell_1 \sqsubseteq \ell_2 \qquad C \vdash \ell_2 \sqsubseteq \ell_3}{C \vdash \ell_1 \sqsubseteq \ell_3} \qquad [C5]\ \frac{C \vdash \ell_1 \sqsubseteq \ell_3 \qquad C \vdash \ell_2 \sqsubseteq \ell_3}{C \vdash \ell_1 \sqcup \ell_2 \sqsubseteq \ell_3}$$

Figure 3: Relabeling rules

$$[S1]\ \frac{C \vdash \tau_1 \leq \tau_2 \quad C \vdash \tau_2 \leq \tau_1}{C \vdash \tau_1\ \texttt{ref} \leq \tau_2\ \texttt{ref}} \qquad [S2]\ \frac{\begin{array}{c} C \vdash \tau_2 \leq \tau_1 \quad C \vdash \tau_1' \leq \tau_2' \\ C \vdash pc_2 \sqsubseteq pc_1 \quad C_2 \vdash C_1 \end{array}}{C \vdash (x{:}\tau_1) \xrightarrow{C_1\,;\,pc_1} \tau_1' \leq (x{:}\tau_2) \xrightarrow{C_2\,;\,pc_2} \tau_2'}$$

$$[S3]\ \frac{C \vdash \tau_1 \leq \tau_2 \quad C \vdash \tau_1' \leq \tau_2' \quad C_1 \vdash C_2}{C \vdash (x{:}\tau_1)[C_1] * \tau_1' \leq (x{:}\tau_2)[C_2] * \tau_2'} \qquad [S4]\ \frac{C \vdash \beta_1 \leq \beta_2 \quad C \vdash \ell_1 \sqsubseteq \ell_2}{C \vdash (\beta_1)_{\ell_1} \leq (\beta_2)_{\ell_2}}$$

Figure 4: Subtyping rules

4.1.4 Subtyping

The subtyping relationship between security types plays an important role in enforcing information flow security. Given two security types $\tau_1 = \beta_{1\ell_1}$ and $\tau_2 = \beta_{2\ell_2}$, suppose τ_1 is a subtype of τ_2, written as $\tau_1 \leq \tau_2$. Then any data of type τ_1 can be treated as data of type τ_2. Thus, data with label ℓ_1 may be treated data with label ℓ_2, which requires $\ell_1 \sqsubseteq \ell_2$.

As described in Section 3.1, the type system keeps track of the set of label constraints that can be used to prove relabeling relationships between labels. Let $C \vdash \ell_1 \sqsubseteq \ell_2$ denote that $\ell_1 \sqsubseteq \ell_2$ can be inferred from the set of constraints C. The inference rules are shown in Figure 3; they are standard and consistent with the lattice properties of labels. Rule (C2) shows that all the constraints in C are assumed to be true. The constraint set C may contain constraints that are inconsistent with the lattice $\mathcal{L}$, such as $H \sqsubseteq L$. Inconsistent constraint sets are harmless because they always indicate dead code, such as expression e_1 in "$\texttt{if}\ H \sqsubseteq L\ \texttt{then}\ e_1\ \texttt{else}\ e_2$".

Since the subtyping relationship depends on the relabeling relationship, the subtyping context also needs to include the C component of the typing context. The inference rules for proving $C \vdash \tau_1 \leq \tau_2$ are the rules shown in Figure 4 plus the standard reflexivity and transitivity rules.

Rules (S1)–(S3) are about subtyping on base types. These rules demonstrate the expected covariance or contravariance. In λ_{DSec}, function types contain two additional components *pc* and C, both of which are contravariant. Suppose the function type $\tau = (x:\tau_1) \xrightarrow{C_1\,;\,pc_1} \tau_1'$ is a subtype of $\tau' = (x:\tau_2) \xrightarrow{C_2\,;\,pc_2} \tau_2'$. Then wherever functions with type τ' can be called, functions with type τ can also be called. This implies two necessary premises. First, wherever C_2 is satisfied, C_1 is also satisfied. This premise is written $C_2 \vdash C_1$, meaning that for any constraint $\ell_1 \sqsubseteq \ell_2$ in C_1, we can derive $C_2 \vdash \ell_1 \sqsubseteq \ell_2$. Second, the premise $pc_2 \sqsubseteq pc_1$ is needed because the *pc* of a function type is an upper bound on the *pc* where the function is applied.

Rule (S4) is used to determine the subtyping on security types. The premise $C \vdash \beta_1 \leq \beta_2$ is natural. The other premise $C \vdash \ell_1 \sqsubseteq \ell_2$ guarantees that coercing data from τ_1 to τ_2 does not violate information flow policies.

$$[\textit{INT}] \quad \Gamma\,;C\,;\,\textit{pc} \vdash n : \texttt{int}_\bot \qquad [\textit{UNIT}] \quad \Gamma\,;C\,;\,\textit{pc} \vdash () : \texttt{unit}_\bot$$

$$[\textit{LABEL}] \quad \Gamma\,;C\,;\,\textit{pc} \vdash k : \texttt{label}_\bot \qquad [\textit{LOC}] \quad \frac{\textit{FV}(\tau) = \emptyset}{\Gamma\,;C\,;\,\textit{pc} \vdash m^\tau : (\tau\ \texttt{ref})_\bot}$$

$$[\textit{JOIN}] \quad \frac{\Gamma\,;C\,;\,\textit{pc} \vdash \ell_1 : \texttt{label}_{\ell_1'} \quad \Gamma\,;C\,;\,\textit{pc} \vdash \ell_2 : \texttt{label}_{\ell_2'}}{\Gamma\,;C\,;\,\textit{pc} \vdash \ell_1 \sqcup \ell_2 : \texttt{label}_{\ell_1' \sqcup \ell_2'}} \qquad [\textit{VAR}] \quad \frac{x{:}\tau \in \Gamma}{\Gamma\,;C\,;\,\textit{pc} \vdash x : \tau}$$

$$[\textit{REF}] \quad \frac{\Gamma\,;C\,;\,\textit{pc} \vdash e : \tau \qquad C \vdash \textit{pc} \sqsubseteq \tau}{\Gamma\,;C\,;\,\textit{pc} \vdash \texttt{ref}^\tau e : (\tau\ \texttt{ref})_\bot} \qquad [\textit{DEREF}] \quad \frac{\Gamma\,;C\,;\,\textit{pc} \vdash e : (\tau\ \texttt{ref})_\ell}{\Gamma\,;C\,;\,\textit{pc} \vdash !e : \tau \sqcup \ell}$$

$$[\textit{ABS}] \quad \frac{\Gamma, x{:}\tau'\,;C'\,;\textit{pc}' \vdash e : \tau}{\Gamma\,;C\,;\,\textit{pc} \vdash \lambda(x{:}\tau')[C'\,;\textit{pc}'].\, e : ((x{:}\tau') \xrightarrow{C'\,;\,\textit{pc}'} \tau)_\bot} \qquad [\textit{ASSIGN}] \quad \frac{\Gamma\,;C\,;\,\textit{pc} \vdash e_1 : (\tau\ \texttt{ref})_\ell \quad \Gamma\,;C\,;\,\textit{pc} \vdash e_2 : \tau \quad C \vdash \textit{pc} \sqcup \ell \sqsubseteq \tau}{\Gamma\,;C\,;\,\textit{pc} \vdash e_1 := e_2 : \texttt{unit}_\bot}$$

$$[\textit{L-APP}] \quad \frac{\begin{array}{c}\Gamma\,;C\,;\,\textit{pc} \vdash e_1 : ((x{:}\texttt{label}_{\ell'}) \xrightarrow{C'\,;\,\textit{pc}'} \tau)_\ell \\ \Gamma\,;C\,;\,\textit{pc} \vdash \ell_2 : \texttt{label}_{\ell'} \quad C \vdash \textit{pc} \sqcup \ell \sqsubseteq \textit{pc}'[\ell_2/x] \\ C \vdash C'[\ell_2/x] \qquad x \in \textit{FV}(\tau) \cup \textit{FV}(C') \cup \textit{FV}(\textit{pc}')\end{array}}{\Gamma\,;C\,;\,\textit{pc} \vdash e_1\ \ell_2 : \tau[\ell_2/x] \sqcup \ell} \qquad [\textit{APP}] \quad \frac{\begin{array}{c}\Gamma\,;C\,;\,\textit{pc} \vdash e_1 : ((x{:}\tau') \xrightarrow{C'\,;\,\textit{pc}'} \tau)_\ell \\ \Gamma\,;C\,;\,\textit{pc} \vdash e_2 : \tau' \qquad C \vdash \textit{pc} \sqcup \ell \sqsubseteq \textit{pc}' \\ C \vdash C' \quad x \notin \textit{FV}(\tau) \cup \textit{FV}(C') \cup \textit{FV}(\textit{pc}')\end{array}}{\Gamma\,;C\,;\,\textit{pc} \vdash e_1\ e_2 : \tau \sqcup \ell}$$

$$[\textit{PROD}] \quad \frac{\begin{array}{c}\Gamma\,;C\,;\,\textit{pc} \vdash v_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash \tau_2 \\ \Gamma\,;C\,;\,\textit{pc} \vdash v_2[v_1/x] : \tau_2[v_1/x] \qquad C \vdash C'[v_1/x]\end{array}}{\Gamma\,;C\,;\,\textit{pc} \vdash (x{=}v_1[C'],\ v_2{:}\tau_2) : ((x{:}\tau_1)[C'] * \tau_2)_\bot} \qquad [\textit{UNPACK}] \quad \frac{\begin{array}{c}\Gamma\,;C\,;\,\textit{pc} \vdash v : ((x{:}\tau_1)[C'] * \tau_2)_\ell \\ \Gamma, x{:}\tau_1 \sqcup \ell, y{:}\tau_2 \sqcup \ell\,;\ C, C'\,;\ \textit{pc} \vdash e : \tau\end{array}}{\Gamma\,;C\,;\,\textit{pc} \vdash \texttt{let}\ (x,y){=}v\ \texttt{in}\ e : \tau}$$

$$[\textit{IF}] \quad \frac{\begin{array}{c}\Gamma\,;C\,;\,\textit{pc} \vdash \ell_i : \texttt{label}_{\ell_i'} \quad i \in \{1,2\} \\ \Gamma\,;\ C, \ell_1 \sqsubseteq \ell_2\,;\ \textit{pc} \sqcup \ell_1' \sqcup \ell_2' \vdash e_1 : \tau \\ \Gamma\,;C\,;\textit{pc} \sqcup \ell_1' \sqcup \ell_2' \vdash e_2 : \tau\end{array}}{\Gamma\,;C\,;\,\textit{pc} \vdash \texttt{if}\ \ell_1 \sqsubseteq \ell_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \sqcup \ell_1' \sqcup \ell_2'} \qquad [\textit{SUB}] \quad \frac{\Gamma\,;C\,;\,\textit{pc} \vdash e : \tau \qquad C \vdash \tau \leq \tau'}{\Gamma\,;C\,;\,\textit{pc} \vdash e : \tau'}$$

Figure 5: Typing rules for the λ_{DSec} language

4.1.5 Typing

The type system of λ_{DSec} prevents illegal information flows and guarantees that well-typed programs have a noninterference property. The typing rules are shown in Figure 5. The notation $\textit{label}(\beta_\ell) = \ell$ is used to obtain the label of a type, and the notations $\ell \sqsubseteq \tau$ and $\tau \sqsubseteq \ell$ are abbreviations for $\ell \sqsubseteq \textit{label}(\tau)$ and $\textit{label}(\tau) \sqsubseteq \ell$, respectively.

The typing context includes a *type assignment* Γ, a set of constraints C and the program-counter label *pc*. Γ is a finite *ordered* list of $x : \tau$ pairs in the order that they came into scope. For a given x, there is at most one pair $x : \tau$ in Γ.

A variable appearing in a type must be a label variable. Therefore, a type τ is well-formed with respect to type assignment Γ, written $\Gamma \vdash \tau$, if Γ maps all the variables in τ to label types. The definition of well-formed labels ($\Gamma \vdash \ell$) is the same. Consider $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$. For any $0 \leq i \leq n$, the type τ_i may only mention label variables that are already in scope: x_1 through x_{i-1}. Therefore, Γ is well-formed if for any $0 \leq i \leq n$, τ_i is well-formed with respect to $x_1 : \tau_1, \ldots, x_{i-1} : \tau_{i-1}$. For example, "$x : \texttt{label}_L, y : \texttt{int}_x$" is well-formed, but "$y : \texttt{int}_x, x : \texttt{label}_L$" is not. A constraint $\ell_1 \sqsubseteq \ell_2$ is well-formed with respect to Γ if both ℓ_1 and ℓ_2 are well-formed with respect to Γ. A typing context "$\Gamma\,;C\,;\textit{pc}$" is well-formed if Γ is well-formed, and *pc* and all the constraints in C are well-formed with respect to Γ.

The typing assertion $\Gamma\,;C\,;\,\textit{pc} \vdash e : \tau$ means that with the type assignment Γ, current program-counter label as *pc*, and the set of constraints C satisfied, expression e has type τ. The assertion $\Gamma\,;C\,;\,\textit{pc} \vdash e : \tau$ is

well-formed if $\Gamma\,;C\,;\mathit{pc}$ is well-formed, and $\Gamma \vdash \tau$.

Rules (INT), (UNIT), (LABEL) and (LOC) are used to check values. Value v has type $\beta_\bot$ if v has base type β. Rule (VAR) is standard: variable x has type $\Gamma(x)$. Rule (JOIN) checks the join of two labels and assigns a result label that is the join of the labels of the operands.

Rule (REF) checks memory allocation operations. If the *pc* label is high, the generated memory location must not be observable to low-security users, which is guaranteed by the premise $C \vdash \mathit{pc} \sqsubseteq \tau$. Rule (DEREF) checks dereference expressions. Since some information about a reference can be learned by knowing its contents, the result of dereferencing a reference with type $(\tau\ \texttt{ref})_\ell$ has type $\tau \sqcup \ell$, where $\tau \sqcup \ell = \beta_{\ell' \sqcup \ell}$ if τ is $\beta_{\ell'}$. Rule (ASSIGN) checks memory update. As in rule (REF), if the updated memory location has type $(\tau\ \texttt{ref})_\ell$, then $C \vdash \mathit{pc} \sqsubseteq \tau$ is required to prevent illegal implicit flows. In addition, the condition $C \vdash \ell \sqsubseteq \tau$ protects the reference that is assigned to. Without the condition, the following code would be well-typed. However, low-security users can learn whether $x \sqsubseteq L$ by observing which of m_1 and m_2 is updated to 0.

$$\lambda(x{:}\texttt{label}_H)[L].\,((\texttt{if}\ x \sqsubseteq L\ \texttt{then}\ m_1^{\texttt{int}_L}\ \texttt{else}\ m_2^{\texttt{int}_L})\ := 0)$$

Rule (ABS) checks function values. The body is checked with the constraint set C' and the program-counter label pc', so the function can only be called at places where C' is satisfied and the *pc* label is not more restrictive than pc'.

Rules (L-APP) and (APP) are used to check application expressions. Consider an application expression $e_1 e_2$, where e_1 has type $((x : \texttt{label}_{\ell'}) \xrightarrow{C'\,;\,\mathit{pc}'} \tau)_\ell$. Rule (L-APP) is used when the occurrences of x do appear in C', pc' or τ. In this case, the type checker needs to use $C'[e_2/x]$, $\mathit{pc}'[e_2/x]$ or $\tau[e_2/x]$, which are well-formed only if e_2 is a label term ℓ_2. In rule (L-APP), the label of $e_1 \ell_2$ is at least as restrictive as ℓ, preventing the result of e_1 from being leaked. The premise $C \vdash C'[\ell_2/x]$ guarantees that $C'[\ell_2/x]$ are satisfied when the function is invoked. The premise $C \vdash \mathit{pc} \sqcup \ell \sqsubseteq \mathit{pc}'[\ell_2/x]$ ensures that the invocation cannot leak the program counter or the function itself through the memory effects of the function. Rule (APP) applies when x does not appear in C', pc' or τ. In this case, the type of e_1 is just a normal function type, so e_1 can be applied to arbitrary terms.

Rule (PROD) is used to check product values. To check v_2, the occurrences of x in v_2 and τ_2 are both replaced by v_1. If v_1 is not a label, then x cannot appear in τ_2. Thus, $\tau_2[v_1/x]$ is always well-formed no matter whether v_1 is a label or not. Rule (UNPACK) checks product destructors straightforwardly. After unpacking the product value, those product label constraints in C' are in scope and used for checking e.

Rule (IF) checks label-test expressions. The constraint $\ell_1 \sqsubseteq \ell_2$ is added into the typing context when checking the first branch e_1. When checking the branches, the program-counter label subsumes the labels of ℓ_1 and ℓ_2 to protect them from implicit flows. The resulting type contains ℓ_1' and ℓ_2' because the result is influenced by the values of ℓ_1 and ℓ_2.

Rule (SUB) is the standard subsumption rule. If τ is a subtype of τ' with the constraints in C satisfied, then any expression of type τ also has type τ'.

This type system satisfies the subject reduction property and the progress property. The proof is standard, so we simply state the theorems here.

Definition 4.1 (Well-typed memory). A memory M is well-typed if for any memory location m^τ in M, $\vdash M(m^\tau) : \tau$.

Theorem 4.1 (Subject reduction). Suppose $\mathit{pc} \vdash e : \tau$, and there exists a well-typed memory M such that $\langle e,\, M\rangle \longmapsto \langle e,\, M'\rangle$, then M' is well-typed, and $\mathit{pc} \vdash e' : \tau$.

Theorem 4.2 (Progress). If $\mathit{pc} \vdash e : \tau$, and M is a well-typed memory such that $\langle e,\, M\rangle$ is a well-formed configuration, then either e is a value or there exists e' and M' such that $\langle e,\, M\rangle \longmapsto \langle e',\, M'\rangle$.

4.2 Noninterference proof

This section outlines a proof that any well-typed program in λ_{DSec} satisfies the noninterference property. (The full proof is given in the appendix.) Consider an expression e in λ_{DSec}. Suppose e has one free variable x, and $x : \tau \vdash e : \mathtt{int}_L$ where $H \sqsubseteq \tau$. Thus, the value of x is a high-security input to e, and the result of e is a low-security output. Then noninterference requires that for all values v of type τ, evaluating $e[v/x]$ in the same memory must generate the same result, if the evaluation terminates. For simplicity, we only consider that results are integers because they can be compared outside the context of λ_{DSec}.

The noninterference property discussed here is *termination insensitive* [20] because $e[v/x]$ is required to generate the same result only if the evaluation terminates. The type system of λ_{DSec} does not attempt to control termination and timing channels. Control of these channels is largely an orthogonal problem. Termination channels can leak at most one bit per run, so they have often been considered acceptable (e.g., [5, 26]). Some recent work [1, 19, 33] partially addresses the control of timing channels.

Let $\longmapsto^*$ denote the transitive closure of the $\longmapsto$ relationship. The following theorem formalizes the claim that the type system of λ_{DSec} enforces noninterference:

Theorem 4.3 (Noninterference). Suppose $x : \tau \vdash e : \mathtt{int}_L$, and $H \sqsubseteq \tau$. Given two arbitrary values v_1 and v_2 of type τ, and an initial memory M, if $\langle e[v_i/x],\, M\rangle \longmapsto^* \langle v_i',\, M_i'\rangle$ for $i \in \{1, 2\}$, then $v_1' = v_2'$.

To prove this noninterference theorem, we adapt the elegant proof technique developed by Pottier and Simonet for an ML-like security-typed language [18] (which did not have dynamic labels). To show that noninterference holds, it is necessary to reason about the executions of two related terms: $e[v_1/x]$ and $e[v_2/x]$. We extend λ_{DSec} with a bracket construct $(e_1 \,|\, e_2)$ that represents alternative expressions that might arise during the evaluation of two programs that differs initially only in v_1 and v_2. Then $e[v_1/x]$ and $e[v_2/x]$ can be incorporated into a single term $e[(v_1 \mid v_2)/x]$ in the extended language λ^2_{DSec}, providing a syntactic way to reason about two executions.

Using λ^2_{DSec}, the noninterference theorem can be proved in three steps:

1. Prove that the evaluation of λ^2_{DSec} adequately represents the execution of two λ_{DSec} terms. Given a λ^2_{DSec} term e, let $\lfloor e \rfloor_1$ and $\lfloor e \rfloor_2$ represent the two λ_{DSec} terms encoded by e. Further, if M maps x to a λ^2_{DSec} term e, then $\lfloor M \rfloor_i$ maps x to $\lfloor e \rfloor_i$ for $i \in \{1, 2\}$. Then the adequacy of λ^2_{DSec} means that $\langle e,\, M\rangle \longmapsto^* \langle v,\, M'\rangle$ holds in λ^2_{DSec} if and only if $\langle \lfloor e \rfloor_i,\, \lfloor M \rfloor_i\rangle \longmapsto^* \langle \lfloor v \rfloor_i,\, M_i'\rangle$ for $i \in \{1, 2\}$ holds in λ_{DSec}.

2. Prove that λ^2_{DSec} satisfies subject reduction: the result of an expression has the same type as the expression. The type system of λ^2_{DSec} gives the bracket $(e_1 \mid e_2)$ a high-security type. Intuitively, e_1 and e_2 are different terms and may produce different results, which must have high-security types and be unobservable to low-security users because otherwise low-security users can distinguish the two executions, violating noninterference.

3. Prove the noninterference theorem: Because $\langle e[v_i/x],\, M\rangle \longmapsto^* \langle v_i',\, M_i'\rangle$ and $e[v_i/x] = \lfloor e[(v_1|v_2)/x] \rfloor_i$ for $i \in \{1, 2\}$, we have $\langle e[(v_1 \mid v_2)/x],\, M\rangle \longmapsto^* \langle v',\, M'\rangle$, where $\lfloor v' \rfloor_i = v_i'$ for $i \in \{1, 2\}$. By the subject reduction theorem, $\vdash v' : \mathtt{int}_L$, which implies that v' is not a bracket construct. Then v' must be an integer n, and $\lfloor v' \rfloor_1 = \lfloor v' \rfloor_2 = n$.

The appendix details the syntax and semantic extensions of λ^2_{DSec} and proves the key subject reduction theorem of λ^2_{DSec}. The major extension to Pottier's proof technique is that the bracket construct must also be applied to labels. Because types may contain bracketed labels, the projection operation also applies to typing environments.

5 Related Work

Dynamic information flow control mechanisms [27, 28] track security labels dynamically and use run-time security checks to constrain information propagation. These mechanisms are transparent to programs, but they cannot prevent illegal implicit flows arising from the control flow paths not taken at run time.

Various general security models [11, 24, 7] have been proposed to incorporate dynamic labeling. Unlike noninterference, these models define what it means for a system to be secure according to a certain relabeling policy, which may allow downgrading labels.

Using static program analysis to check information flow was first proposed by Denning and Denning [5]; later work phrased the analysis as type checking (e.g., [17]). Noninterference was later developed as a more semantic characterization of security [8], followed by many extensions. Volpano, Smith and Irvine [26] first showed that type systems can be used to enforce noninterference, and proved a version of noninterference theorem for a simple imperative language, starting a line of research pursuing the noninterference result for more expressive security-typed languages. Heintze and Riecke [10] proved the noninterference theorem for the SLam calculus, a purely functional language. Zdancewic and Myers [32] investigated a secure calculus with first-class continuations and references. Pottier and Simonet [18] considered an ML-like functional language, and demonstrated the innovative proof technique that is used in this paper to reduce the proof of noninterference to a proof of subject reduction. Banerjee and Naumann [3] proved a noninterference result for a Java-like language. A more complete survey of language-based information-flow techniques can be found in [20].

The Jif language [14, 16] extends Java with a type system for analyzing information flow, and aims to be a practical language for developing secure applications. However, there is not yet a noninterference proof for the type system of Jif, because of its complexity. This work is inspired by the dynamic label mechanism of Jif, although the dynamic label mechanism in Jif-DX and λ_{DSec} is more expressive.

Concurrent to this work, Tse and Zdancewic proved a noninterference result for a security-typed lambda calculus ($\lambda_{\mathtt{DP}}$) with dynamic principals [25]. Our work is more general in the sense that it can be applied to label models that do not involve principals. In addition, λ_{DSec} has more computational power than $\lambda_{\mathtt{DP}}$ because λ_{DSec} supports references, which can used to encode recursive functions. The type system of $\lambda_{\mathtt{DP}}$ uses *singleton types* [2] to enforce that every dynamic principal term has a static counterpart, and the dynamism of security policies is captured by the principal hierarchy and a delegation mechanism. It is not clear that this approach can be easily generalized to dynamic labels.

Other work [30, 29] has used dependent type systems to specify complex program invariants and to statically catch program errors considered run-time errors by traditional type systems. This work also makes a trade-off between expressive power and practical type checking.

6 Conclusions

This paper makes two contributions: first, it presents the Jif-DX language that extends the Jif programming model with better support for dynamic labels. The extensions proposed in Jif-DX make it easier to write programs manipulating dynamic labels and can reduce the number of run-time label checks. The key new element is a restricted form of label constraints that is expressive enough for implementing run-time security checks, yet suitable for static type checking. Label constraints also make it possible to encode previous mandatory access control mechanisms that support dynamically changing labels.

Second, this paper formalizes computation and static checking of dynamic labels in the type system of a core language λ_{DSec} and proves a noninterference result: well-typed programs have the noninterference property. The language λ_{DSec} is the first language supporting general dynamic labels whose type system provably enforces noninterference.

An important direction for future work is to investigate the interaction between dynamic labels and parametric polymorphism.

Acknowledgements

The authors would like to thank Greg Morrisett, Steve Zdancewic and Amal Ahmed for their insightful suggestions. Steve Chong, Nate Nystrom, and Michael Clarkson also helped improve the presentation of this work.

References

[1] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, January 2000.

[2] David Aspinall. Subtyping with singleton types. In *Computer Science Logic (CSL), Kazimierz, Poland*, pages 1–15. Springer-Verlag, 1994.

[3] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2002.

[4] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[5] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[6] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD (The Orange Book) edition, December 1985.

[7] Simon Foley, Li Gong, and Xiaolei Qian. A security model of dynamic labeling providing a tiered approach to verification. In *IEEE Symposium on Security and Privacy*, pages 142–154, Oakland, CA, 1996. IEEE Computer Society Press.

[8] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996. ISBN 0-201-63451-1.

[10] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, January 1998.

[11] John McLean. The algebra of security. In *IEEE Symposium on Security and Privacy*, pages 2–7, Oakland, California, 1988.

[12] Catherine Meadows. Policies for dynamic upgrading. In *Database Security, IV: Status and Prospects*, pages 241–250. North Holland, 1991.

[13] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, Cambridge, Massachusetts, 1996.

[14] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[15] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[16] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2003.

[17] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, September 1995.

[18] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[19] Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *Proceedings of the 9th International Static Analysis Symposium*, volume 2477 of *LNCS*, Madrid, Spain, September 2002. Springer-Verlag.

[20] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[21] Ravi S. Sandhu and Sushil Jajodia. Honest databases that can keep secrets. In *Proceedings of the 14th National Computer Security Conference*, Washington, DC, 1991.

[22] Vincent Simonet. Fine-grained information flow analysis for a lambda-calculus with sum types. In *Proc. 15th IEEE Computer Security Foundations Workshop*, pages 223–237, June 2002.

[23] Vincent Simonet. An extension of HM(X) with first class existential and universal data-types. In *Proc. 8th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 39–50, Uppsala, Sweden, August 2003.

[24] Ian Sutherland, Stanley Perlo, and Rammohan Varadarajan. Deducibility security with dynamic level assignments. In *Proc. 2nd IEEE Computer Security Foundations Workshop*, Franconia, NH, June 1989.

[25] Stephen Tse and Steve Zdancewic. Dynamic principals in security-typed languages. In *IEEE Symposium on Security and Privacy*, Oakland, CA, 2004 (To appear).

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

[27] Clark Weissman. Security controls in the ADEPT-50 time-sharing system. In *AFIPS Conference Proceedings*, volume 35, pages 119–133, 1969.

[28] John P. L. Woodward. Exploiting the dual nature of sensitivity labels. In *IEEE Symposium on Security and Privacy*, pages 23–30, Oakland, California, 1987.

[29] Hongwei Xi. Imperative programming with dependent types. In *Proceedings of 15th Symposium on Logic in Computer Science*, Santa Barbara, June 2000.

[30] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 214–227, San Antonio, TX, January 1999.

[31] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Nova Scotia, Canada, June 2001.

[32] Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2–3):209–234, September 2002.

[33] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, Pacific Grove, California, June 2003.

A Subject Reduction Proof

As described in Section 4.2, the noninterference result for λ_{DSec} is proved by extending the language to a new language λ^2_{DSec} that includes the special bracket construct. Then the subject reduction property for λ^2_{DSec} implies the noninterference property for λ_{DSec}. The appendix details the syntax and semantic extensions of λ^2_{DSec} and proves the key subject reduction theorem.

A.1 Syntax extensions

The syntax extensions of λ^2_{DSec} include the bracket constructs and a new value `void` that can have any type. A λ^2_{DSec} memory encodes two λ_{DSec} memories, which may have distinct domains. The bindings of the form $m^\tau \mapsto (v \mid \texttt{void})$ and $m^\tau \mapsto (\texttt{void} \mid v)$ represent situations where m^τ is bound within only one of the two λ_{DSec} memories.

$$\begin{array}{lcl} \ell & ::= & \ldots \mid (\ell \mid \ell) \\ v & ::= & \ldots \mid (v \mid v) \mid \texttt{void} \\ e & ::= & \ldots \mid (e \mid e) \end{array}$$

The bracket constructs cannot be nested, so the subterms of a bracket construct must be λ_{DSec} terms or `void`. Given a λ^2_{DSec} expression e, let $\lfloor e \rfloor_1$ and $\lfloor e \rfloor_2$ represent the two λ_{DSec} terms that e encodes. The projection functions satisfy $\lfloor (e_1 | e_2) \rfloor_i = e_i$ and are homomorphisms on other expression forms. In addition, $(e_1 \mid e_2)[v/x]$, the capture-free substitution of v for x in $(e_1 \mid e_2)$, must use the corresponding projection of v in each branch: $(e_1 \mid e_2)[v/x] = (e_1[\lfloor v \rfloor_1/x] \mid e_2[\lfloor v \rfloor_2/x])$.

In λ^2_{DSec}, labels can be bracket constructs, and types may contain bracketed labels. Thus, the projection operation can be applied to labels, types, type assignments, and label constraints. Similarly, the projection functions are homomorphisms on these typing constructs. For example, $\lfloor \texttt{int}_{(L \mid H)} \rfloor_1 = \texttt{int}_L$, and $\lfloor x : \tau, y : \tau' \rfloor_1 = x : \lfloor \tau \rfloor_1, y : \lfloor \tau' \rfloor_1$.

The following relabeling rule is added to reason about relabeling relationship between bracketed labels:

$$\frac{\lfloor C \rfloor_1 \vdash \lfloor \ell_1 \rfloor_1 \sqsubseteq \lfloor \ell_2 \rfloor_1 \qquad \lfloor C \rfloor_2 \vdash \lfloor \ell_1 \rfloor_2 \sqsubseteq \lfloor \ell_2 \rfloor_2}{C \vdash \ell_1 \sqsubseteq \ell_2}$$

Since a λ^2_{DSec} term effectively encodes two λ_{DSec} terms, the evaluation of a λ^2_{DSec} term can be projected into two λ_{DSec} evaluations. An evaluation step of a bracket expression $(e_1 \mid e_2)$ is an evaluation step of either e_1 or e_2. and e_1 or e_2 can only access the corresponding projection of the memory. Thus, the configuration of λ^2_{DSec} has an index $i \in \{\bullet, 1, 2\}$ that indicates whether the term to be evaluated is a subterm of a bracket expression, and if so which branch of a bracket the term belongs to. For example, the configuration $\langle e,\, M \rangle_1$ means that e belongs to the first branch of a bracket, and e can only access the first projection of M. We write "$\langle e,\, M \rangle$" for "$\langle e,\, M \rangle_\bullet$", which means e does not belong to any bracket.

A.2 Operational semantics

The operational semantics of λ^2_{DSec} is shown in Figure 6. It is based on the semantics of λ_{DSec} and contains some new evaluation rules (E10–E14) for manipulating bracket constructs. Rules (E2)–(E4) are modified to access the memory projection corresponding to index i. The rest of the rules in Figure 2 are adapted to λ^2_{DSec} by indexing each configuration with i. The following two lemmas state that the operational semantics of λ^2_{DSec} is adequate to encode the execution of two λ_{DSec} terms. Their proof is straightforward.

Lemma A.1 (Soundness). If $\langle e,\, M \rangle \longmapsto \langle e',\, M' \rangle$, then $\langle \lfloor e \rfloor_i,\, \lfloor M \rfloor_i \rangle \longmapsto \langle \lfloor e' \rfloor_i,\, \lfloor M' \rfloor_i \rangle$ for $i \in \{1, 2\}$.

Lemma A.2 (Completeness). If $\langle \lfloor e \rfloor_i,\, \lfloor M \rfloor_i \rangle \longmapsto^* \langle v_i,\, M_i' \rangle$ for $i \in \{1, 2\}$, then there exists a configuration $\langle v,\, M' \rangle$ such that $\langle e,\, M \rangle \longmapsto^* \langle v,\, M' \rangle$.

The type system of λ^2_{DSec} includes all the typing rules in Figure 5 and has two additional rules, one for typing `void`, the other for typing bracket constructs.

$$[\textit{VOID}] \qquad \Gamma\,;C\,;\,\textit{pc} \vdash \texttt{void} : \tau$$

$$[\textit{BRACKET}] \quad \frac{\lfloor \Gamma \rfloor_1\,;\lfloor C \rfloor_1\,;\lfloor \textit{pc}' \rfloor_1 \vdash e_1 : \lfloor \tau \rfloor_1 \quad \lfloor \Gamma \rfloor_2\,;\lfloor C \rfloor_2\,;\lfloor \textit{pc}' \rfloor_2 \vdash e_2 : \lfloor \tau \rfloor_2 \quad H \sqcup \textit{pc} \sqsubseteq \textit{pc}' \quad H \sqsubseteq \tau}{\Gamma\,;C\,;\,\textit{pc} \vdash (e_1 \mid e_2) : \tau}$$

A.3 Subject reduction

The proof of subject reduction starts with some lemmas about projection and substitution.

Lemma A.3 (Label Projection). If $C \vdash \ell_1 \sqsubseteq \ell_2$, then $\lfloor C \rfloor_i \vdash \lfloor \ell_1 \rfloor_i \sqsubseteq \lfloor \ell_2 \rfloor_i$ for $i \in \{1, 2\}$.

Proof. By induction on the derivation of $C \vdash \ell_1 \sqsubseteq \ell_2$. □

[*E2*] $\langle !m^\tau,\, M\rangle_i \longmapsto \langle \mathtt{read}_i\, M(m^\tau),\, M\rangle_i$

[*E3*] $$\frac{m = \mathit{newloc}(M)}{\langle \mathtt{ref}^\tau v,\, M\rangle_i \longmapsto \langle m^\tau,\, M[m^\tau \mapsto \mathtt{new}_i\, v]\rangle_i}$$

[*E4*] $\langle m^\tau := v,\, M\rangle_i \longmapsto \langle (),\, M[m^\tau \mapsto \mathtt{update}_i\, M(m^\tau)\, v]\rangle_i$

[*E10*] $$\frac{\langle e_i,\, M\rangle_i \longmapsto \langle e_i',\, M'\rangle_i \quad e_j = e_j' \qquad \{i,j\} = \{1,2\}}{\langle (e_1 \mid e_2),\, M\rangle \longmapsto \langle (e_1' \mid e_2'),\, M'\rangle}$$

[*E11*] $\langle (v_1 \mid v_2)v,\, M\rangle \longmapsto \langle (v_1\lfloor v\rfloor_1 \mid v_2\lfloor v\rfloor_2),\, M\rangle$

[*E12*] $\langle (v_1 \mid v_2) := v,\, M\rangle \longmapsto \langle (v_1 := \lfloor v\rfloor_1 \mid v_2 := \lfloor v\rfloor_2),\, M\rangle$

[*E13*] $\langle !(v_1 \mid v_2),\, M\rangle \longmapsto \langle (!v_1 \mid !v_2),\, M\rangle$

[*E14*] $$\langle \mathtt{if}\ v_1 \sqsubseteq v_2\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2,\, M\rangle \longmapsto \langle (\mathtt{if}\ \lfloor v_1\rfloor_1 \sqsubseteq \lfloor v_2\rfloor_1\ \mathtt{then}\ \lfloor e_1\rfloor_1\ \mathtt{else}\ \lfloor e_2\rfloor_1 \mid \mathtt{if}\ \lfloor v_1\rfloor_2 \sqsubseteq \lfloor v_2\rfloor_2\ \mathtt{then}\ \lfloor e_1\rfloor_2\ \mathtt{else}\ \lfloor e_2\rfloor_2), M\rangle$$
if $v_1 = (v \mid v')$ or $v_2 = (v \mid v')$

[Auxiliary functions]

$$\begin{array}{lll} \mathtt{new}_\bullet\, v = v & \mathtt{update}_\bullet\, vv' = v' & \mathtt{read}_\bullet\, v = v \\ \mathtt{new}_1\, v = (v \mid \mathtt{void}) & \mathtt{update}_1\, vv' = (v' \mid \lfloor v\rfloor_2) & \mathtt{read}_1\, v = \lfloor v\rfloor_1 \\ \mathtt{new}_2\, v = (\mathtt{void} \mid v) & \mathtt{update}_2\, vv' = (\lfloor v\rfloor_1 \mid v') & \mathtt{read}_2\, v = \lfloor v\rfloor_2 \end{array}$$

Figure 6: Small-step operational semantics of λ^2_{DSec}

Lemma A.4 (Constraint Reduction). If $\Gamma\,;C, \ell_1 \sqsubseteq \ell_2\,;\mathit{pc} \vdash e : \tau$ and $C \vdash \ell_1 \sqsubseteq \ell_2$, then $\Gamma\,;C\,;\ \mathit{pc} \vdash e : \tau$.

Proof. By induction on the derivation of $\Gamma\,;C, \ell_1 \sqsubseteq \ell_2\,;\mathit{pc} \vdash e : \tau$. □

Lemma A.5 (Projection). If $\Gamma\,;C\,;\ \mathit{pc} \vdash e : \tau$, then $\lfloor\Gamma\rfloor_i\,;\lfloor C\rfloor_i\,;\lfloor \mathit{pc}\rfloor_i \vdash \lfloor e\rfloor_i : \lfloor\tau\rfloor_i$, for $i \in \{1,2\}$.

Proof. By induction on the derivation of $\Gamma\,;C\,;\ \mathit{pc} \vdash e : \tau$, and using the label projection lemma. □

Lemma A.6 (Store Access). Let i be in $\{\bullet, 1, 2\}$. Suppose $\mathit{pc} \vdash v : \tau$ and $\mathit{pc} \vdash v' : \tau$. In addition, $i \in \{1,2\}$ implies $H \sqsubseteq \tau$. Then $\mathit{pc} \vdash \mathtt{read}_i\, v : \lfloor\tau\rfloor_i$, $\mathit{pc} \vdash \mathtt{new}_i\, v : \tau$ and $\mathit{pc} \vdash \mathtt{update}_i\, vv' : \tau$.

Proof. By the definition of the functions `read`, `new` and `update` in Figure 6, by the projection lemma, and rules (VOID) and (BRACKET). □

Lemma A.7 (Substitution). If $x : \tau', \Gamma\,;C\,;\mathit{pc} \vdash\ e\ :\ \tau$, and $\vdash\ v\ :\ \tau'$, then $\Gamma[v/x]\,;C[v/x]\,;\mathit{pc}[v/x] \vdash e[v/x] : \tau[v/x]$.

Proof. By induction on the derivation of $x{:}\tau', \Gamma\,;C\,;\mathit{pc} \vdash e : \tau$. □

Theorem A.1 (Subject Reduction). Suppose $\mathit{pc} \vdash e : \tau$, memory M is well-typed, $\langle e,\, M\rangle_i \longmapsto \langle e',\, M'\rangle_i$, and $i \in \{1,2\}$ implies $H \sqsubseteq \mathit{pc}$. Then $\mathit{pc} \vdash e' : \tau$, and M' is also well-typed.

Proof. By induction on the derivation of $\langle e,\ M\rangle_i \longmapsto \langle e',\ M'\rangle_i$. Without loss of generality, we assume that the last step of the derivation of $pc \vdash e : \tau$ does not use the rule (SUB). Here we just show seven cases: (E3), (E5), (E6), (E8), (E10), (E11) and (E14). The rest of evaluation rules are treated similarly.

- Case (E3). e is $\texttt{ref}^{\tau'}\ v$, and τ is $(\tau'\ \texttt{ref})_\perp$. Then e' is $m^{\tau'}$. By (LOC), $pc \vdash e' : (\tau'\ \texttt{ref})_\perp$. By Lemma A.6, $pc \vdash \texttt{new}_i v : \tau'$. Thus, $M[m^{\tau'} \mapsto \texttt{new}_i v]$ is well-typed.

- Case (E5). e is $(\lambda(x:\tau')[C';pc'].\,e')v$. Then $pc \vdash \lambda(x:\tau')[C';pc'].\,e' : ((x:\tau'') \xrightarrow{C'';pc''} \tau_1)_\ell$, and $pc \vdash v : \tau''$, and $\vdash C''[v/x]$. By rules (APP) and (L-APP), $\tau = \tau_1[v/x] \sqcup \ell$, and $pc \sqsubseteq pc''[v/x]$. By rules (ABS) and (SUB), $x : \tau'; C'; pc' \vdash e' : \tau_1$, and $\vdash \tau'' \leq \tau'$, $\vdash pc'' \sqsubseteq pc'$, and $C'' \vdash C'$. Therefore, $\vdash C'[v/x]$, and $pc \sqsubseteq pc'[v/x]$. By the substitution lemma, $C'[v/x]; pc'[v/x] \vdash e'[v/x] : \tau_1[v/x]$. By Lemma A.4, $pc'[v/x] \vdash e'[v/x] : \tau_1[v/x]$. Since $pc \sqsubseteq pc'[v/x]$ and $\tau_1[v/x] \sqsubseteq \tau$, we have $pc \vdash e'[v/x] : \tau$.

- Case (E6). By rule (IF), $k_1 \sqsubseteq k_2; pc \vdash e_1 : \tau$. By Lemma A.4 and $\mathcal{L} \models k_1 \sqsubseteq k_2$, we have $pc \vdash e_1 : \tau$.

- Case (E8). e is $\texttt{let}\ (x,y) = (x = v_1[C],\ v_2 : \tau_2)\ \texttt{in}\ e'$. By rule (UNPACK), $pc \vdash (x = v_1[C],\ v_2 : \tau_2) : ((x:\tau_1)[C] * \tau_2)_\ell$, and $x : \tau_1 \sqcup \ell, y : \tau_2 \sqcup \ell; pc \vdash e' : \tau$. By rule (PROD), $pc \vdash v_1 : \tau_1$, and $pc \vdash v_2[v_1/x] : \tau_2[v_1/x]$, and $\vdash C[v_1/x]$. Using the substitution lemma twice, we get $C[v_1/x]; pc \vdash e'[v_1/x][v_2[v_1/x]/y] : \tau[v_1/x][v_2[v_1/x]/y]$. It is easy to show that $e'[v_1/x][v_2[v_1/x]/y] = e'[v_2/y][v_1/x]$. According to rule (UNPACK), $x, y \notin FV(\tau)$. Thus, $\tau[v_1/x][v_2[v_1/x]/y] = \tau$. In addition, we have $\vdash C[v_1/x]$. Therefore, $pc \vdash e[v_1/x][v_2/y] : \tau$.

- Case (E10). e is $(e_1 \mid e_2)$. Without loss of generality, assume $\langle e_1,\ M\rangle_1 \longmapsto \langle e_1',\ M'\rangle_1$ and $e_2 = e_2'$. By rule (BRACKET), $H \sqsubseteq pc$, and $\lfloor pc \rfloor_1 \vdash e_1 : \lfloor \tau \rfloor_1$. $H \sqsubseteq pc$ implies $H \sqsubseteq \lfloor pc \rfloor_1$. By induction, $\lfloor pc \rfloor_1 \vdash e_1' : \lfloor \tau \rfloor_1$, and M' is well-typed. Using rule (BRACKET), we can get $pc \vdash (e_1' \mid e_2') : \tau$.

- Case (E11). e is $(v_1 \mid v_2)v$. By (APP) and (L-APP), $pc \vdash (v_1 \mid v_2) : ((x:\tau') \xrightarrow{C';pc'} \tau'')_\ell$, and $pc \vdash v : \tau'$. Then $\tau = \tau''[v/x] \sqcup \ell$. In addition, $pc \sqcup \ell \sqsubseteq pc'$. By (BRACKET), $H \sqsubseteq \ell$, which implies $H \sqsubseteq pc'$. By Lemma A.5, $\lfloor pc \rfloor_i \vdash v_i : ((x : \lfloor \tau' \rfloor_i) \xrightarrow{\lfloor C' \rfloor_i;\lfloor pc' \rfloor_i} \lfloor \tau \rfloor_i)_{\lfloor \ell \rfloor_i}$, and $\lfloor pc \rfloor_i \vdash \lfloor v \rfloor_i : \lfloor \tau' \rfloor_i$, which imply $\lfloor pc \rfloor_i \vdash v_i \lfloor v \rfloor_i : \lfloor \tau \rfloor_i$. According to (APP) and (L-APP), a well-typed application expression $e_1 e_2$ can be type-checked with the pc component of the type of e_1 in the typing context. Therefore, $\lfloor pc' \rfloor_i \vdash v_i \lfloor v \rfloor_i : \lfloor \tau \rfloor_i$. Since $H \sqsubseteq pc'$, we can apply (BRACKET) to get $pc \vdash (v_1 \lfloor v \rfloor_1 \mid v_2 \lfloor v \rfloor_2) : \tau$.

- Case (E14). e is $\texttt{if}\ v_1 \sqsubseteq v_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, and there exists $j \in \{1,2\}$ such that $v_j = (v \mid v')$. Suppose $pc \vdash v_i : \texttt{label}_{\ell_i}$ for $i \in \{1,2\}$. Since v_j is a bracket construct, $H \sqsubseteq \ell_j$. By (IF), both e_1 and e_2 are type-checked with $pc \sqcup \ell_1 \sqcup \ell_2$ in the typing context. Thus, we can get $pc \sqcup \ell_1 \sqcup \ell_2 \vdash e : \tau$. By Lemma A.5, $\lfloor pc \sqcup \ell_1 \sqcup \ell_2 \rfloor_i \vdash \lfloor e \rfloor_i : \lfloor \tau \rfloor_i$. $H \sqsubseteq \ell_j$ implies $H \sqsubseteq \lfloor pc \sqcup \ell_1 \sqcup \ell_2 \rfloor_i$. Applying (BRACKET), we get $pc \vdash (\lfloor e \rfloor_1 \mid \lfloor e \rfloor_2) : \tau$.

□