

Efficient Type Inference for Secure Information Flow*

Katia Hristova,[†] Tom Rothamel, Yanhong A. Liu, and Scott D. Stoller

October 15, 2007

Abstract

This paper describes the design, analysis, and implementation of an efficient algorithm for information flow analysis expressed using a type system. Given a program and an environment of security classes for information accessed by the program, the algorithm checks whether the program is well typed, which ensures that no information of higher security classes flows into locations with lower security classes, by inferring the highest or lowest security class, as appropriate, for each program node. We express the analysis as a set of extended Datalog rules based on the typing and subtyping rules, and we use a systematic method to generate specialized algorithms and data structures directly from the rules. The generated implementation traverses the program multiple times and uses a combination of linked and indexed data structures to represent program nodes, environments, and types. The time complexity of the algorithm is linear in the size of the input program, times the height of the lattice of security classes, plus an overhead for preprocessing the lattice. This complexity is confirmed through our prototype implementation and experimental evaluation on code generated from high-level specifications for real systems.

1 Introduction

Protection of the confidentiality and privacy of data is becoming increasingly important. In addition to controlling direct access to information, it is also essential to control information flow, especially in untrusted code. Static analysis of information flow in programs allows fine-grained control without runtime overhead.

Denning's pioneering work [10, 11] on this subject proposed a lattice model that can be used to verify secure information flow in programs. In this model, security classes are ordered in a lattice, and program variables and data are each assigned a security class. The basic security requirement is the absence of information flow from higher to lower security classes. Security classes can indicate the level of secrecy or the level of integrity of data.

Based on Denning's lattice model of information flow analysis, several type-based approaches have been developed [27, 24, 19, 1, 5]. In these works, formal systems of typing rules are used to reason about information flow properties of programs. Volpano, Irvine and Smith [27] were the first to formulate Denning's lattice model as a type system and show that it is sound. Information flow is guaranteed to be secure for a program if the program type-checks correctly.

A decade after this type-based approach was introduced, the first type inference algorithm for information flow types based on the first type system [27] was developed by Deng and Smith [9]. The algorithm has a quadratic running time in the size of the program.

*This work was supported in part by NSF under grants CCR-0306399 and CCF-0613913, and ONR under grant N00014-04-1-0722.

[†]Corresponding author. Computer Science Department, Stony Brook University, Stony Brook, NY 11794-4400. Email katia@cs.sunysb.edu.

This paper presents the design, analysis, and implementation of an efficient linear-time algorithm for information flow analysis expressed using such a type system [27, 24, 9]. Given a program and an environment specifying security classes for information accessed by the program, our algorithm checks whether the program is well typed, i.e., there is no information of higher security classes flowing into places of lower security classes, by inferring the highest or lowest security class as appropriate for each program node. We express the analysis as a set of extended Datalog rules based on the typing and subtyping rules, and we use a systematic method to generate specialized algorithms and data structures directly from the extended Datalog rules. Datalog is a database query language based on the logic programming paradigm [8, 2]. Our extended Datalog rules are Datalog rules with negation and external functions. The method described in [15] is used to generate specialized algorithms and data structures and complexity formulas for the extended Datalog rules. Given a program and an environment of security types, the algorithm infers minimum or maximum security types, as appropriate, for each program node, such that the program type checks correctly, or reports that the program is untypable. The algorithm traverses the program top-down multiple times to infer minimum expression types, and then traverses the program bottom-up once to infer maximum command types. The generated implementation uses a combination of linked and indexed data structures to represent program nodes, environments, and types. The implementation employs an incremental approach that considers one program node at a time. The running time is optimal for the set of rules we use to specify type inference, in the sense that each combination of instantiations of hypotheses is considered once in constant time.

The time complexity of the algorithm is linear in the size of the input program, times the height of the lattice of security classes, plus a small overhead for preprocessing the lattice.

The main contributions of this paper are:

- The first linear time algorithm for efficient type inference for information flow analysis based on a formal type system [27, 24, 9]. The algorithm is presented with complete pseudocode and data structures.
- A novel implementation strategy for type inference for secure information flow types. The strategy combines an intuitive specification of type inference expressed in extended Datalog rules, and a systematic method for deriving efficient algorithms and data structures from the rules.
- Precise and automated time complexity analysis for type inference for secure information flow types. The time complexity is calculated directly from the rules, reflecting the complexities of implementation back into the rules.
- Experimental evaluation of a prototype implementation of our algorithm on code generated from high-level specifications for real systems. The experimental results confirm our complexity analysis.

The rest of this paper is organized as follows. Section 2 reviews the lattice model of analyzing information flow in programs and the type system we consider, and defines the problem of type inference for secure information flow. Section 3 expresses type inference in extended Datalog rules, describes generation of an efficient algorithm and data structure from the extended Datalog rules, and discusses informative error reporting. Section 4 presents the time complexity analysis for the generated algorithm. Section 5 describes experimental results. Section 6 discusses related work and concludes.

2 A Type System for Secure Information Flow

This section reviews the lattice model of information flow [10, 11], and a type system based on it [27].

2.1 Lattice model of secure information flow

In the lattice model of information flow [11, 10] security classes form a lattice, denoted by (SC, \leq) , comprising a finite set SC of security classes, and a partial order \leq . A *security class* is an indication of (i) the level of *secrecy* of the data — how confidential the data is, (ii) the level of *integrity* of the data — how trusted the data is, or (iii) a combination of these two properties. Every program variable is associated with a security class. The security classes of variables are determined statically and do not vary at run time. Every program node is associated with a *certification condition* — a condition relating security classes of neighboring nodes that checks whether the information flow in the node is secure.

Information is considered to *flow* from variable $v1$ into variable $v2$ whenever the value stored in $v1$ affects the value stored in $v2$. Information flow may be explicit or implicit. An *explicit flow* results from assigning the value of a variable to another variable. *Implicit flows* reflect control dependencies. For example, an implicit flow exists from the value of a conditional guard to the branches of the conditional. For example, in the following `if`-statement:

```
if a=0 then b:=1 else b:=0
```

there is an implicit flow from variable `a` to variable `b`, since after the statement has been executed, by the value of variable `b` we can determine whether the value of `a` is 0.

The *flow relation* \rightarrow is a binary relation on security classes that indicates the permitted information flows. For security classes x and y , if $x \rightarrow y$ then flow from variables of class x to variables of class y are permitted and called *secure flows*. In the lattice model, the flow relation is: $x \rightarrow y$ if $x \leq y$.

The lattice model of information flow makes it possible to check conditions on both explicit and implicit information flows by checking the certification conditions on program constructs.

2.2 Type system for secure flow analysis

The type system for secure information flow [27, 9] is based on Denning's lattice model of information flow. The type system guarantees that explicit and implicit flows are secure.

The security types are assumed to form a partial order, denoted by \leq . The strict order induced by the partial order \leq is denoted by $<$. The partial order relation \leq is extended to a *subtype* relation, denoted by \sqsubseteq ; the strict order induced by the partial order \sqsubseteq is denoted by \subset .

Two levels of types are used:

- *data types*, denoted by τ , and are the security classes in the lattice;
- *phrase types*, denoted by ρ , include (i) data types τ , given to expressions; (ii) variable types τ *var* given to variables; (iii) command types τ *cmd* given to commands; and (iv) array types $\tau1$ *arr* $\tau2$ given to arrays.

A variable of type τ *var* stores information whose security class is type τ or lower. A command of type τ *cmd* contains assignments only to variables of type τ or higher. An array of type $\tau1$ *arr* $\tau2$ contains data of security type $\tau1$ and has length of security type $\tau2$. Every array type is subject to the constraint $\tau2 \leq \tau1$. Intuitively, an array's contents include its length, so the security level of the

$$\begin{array}{l}
\text{(BASE)} \quad \frac{r \leq r1}{\vdash r \subseteq r1} \\
\text{(REFLEX)} \quad \vdash \rho \subseteq \rho \\
\text{(TRANS)} \quad \frac{\vdash \rho \subseteq \rho1, \vdash \rho1 \subseteq \rho2}{\vdash \rho \subseteq \rho2} \\
\text{(CMD)}^- \quad \frac{\vdash \rho \subseteq \rho1}{\vdash \rho1 \text{ cmd} \subseteq \rho \text{ cmd}} \\
\text{(SUBTYPE)} \quad \frac{\lambda; \gamma \vdash p : \rho}{\vdash \rho \subseteq \rho1} \\
\lambda; \gamma \vdash p : \rho1
\end{array}$$

Figure 1: Subtyping rules.

contents should be at least as high as that of the length. To ensure that this condition holds for all global arrays, we need to check that for all locations l , if the type of l has the form $\tau1 \text{ arr } \tau2$, then $\tau2 \leq \tau1$.

A *phrase* is an expression or a command generated by the following grammar:

$$\begin{array}{l}
\text{expression } e ::= x \mid l \mid n \mid e1 + e2 \mid e1 - e2 \mid \\
\quad e1 = e2 \mid e1 < e2 \mid a[e1] \mid a.\text{length} \\
\text{command } c ::= e1 := e2 \mid \\
\quad c1; c2 \mid \\
\quad \text{if } e \text{ then } c1 \text{ else } c2 \mid \\
\quad \text{while } e \text{ do } c \mid \\
\quad \text{letid } x := e \text{ in } c \mid \\
\quad a[e1] := e2 \mid \text{allocate } a[e1]
\end{array}$$

Expressions include identifiers x , locations l , integer literals n , arithmetic expressions, and array expressions. Commands of the forms shown above are, respectively, assignments, compositions, conditional commands, local variable (i.e. identifier) declarations, array assignment, and array allocation. The array allocation command `allocate $a[e1]$` allocates an array a of length $e1$.

Typing judgments are of the form $\lambda; \gamma \vdash p : \rho$, where γ is a mapping of identifiers to security types and λ is a mapping of locations to security types. The meaning of this typing judgment is that phrase p has type ρ , if identifiers and locations in p have security types as assigned in γ and λ .

$\gamma[x : \rho]$ denotes a modification of γ that assigns type ρ to identifier x and leaves other identifier-type mappings in γ unchanged.

A typing rule has the form:

$$\frac{J_1 \ J_2 \ \dots \ J_n}{J_{n+1}}$$

where J_i 's are typing judgments. The typing judgments above the line are *hypotheses*, and the typing judgment below the line is the *conclusion*. The rule infers the typing judgment in its conclusion, if all its hypotheses hold. A judgement holds if it is an axiom or can be inferred by some typing rule.

(LITERAL)	$\lambda; \gamma \vdash n : \tau$
(ID)	$\lambda; \gamma \vdash x : \tau \text{ var if } \gamma(x) = \tau \text{ var}$
(LOC)	$\lambda; \gamma \vdash l : \tau \text{ var if } \lambda(l) = \tau$
(R-VAL)	$\frac{\lambda; \gamma \vdash e : \tau \text{ var}}{\lambda; \gamma \vdash e : \tau}$
(ARRLEN)	$\frac{\lambda(a) = \tau 1 \text{ arr } \tau 2}{\lambda; \gamma \vdash a.\text{length} : \tau 2}$
(ARRACCESS)	$\frac{\lambda(a) = \tau 1 \text{ arr } \tau 2, \lambda; \gamma \vdash e : \tau 3}{\lambda; \gamma \vdash a[e] : \tau 1 \vee \tau 3}$
(ARITH)	$\frac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma \vdash e 1 : \tau}{\lambda; \gamma \vdash e + e 1 : \tau}$
(ASSIGN)	$\frac{\lambda; \gamma \vdash e : \tau \text{ var} \quad \lambda; \gamma \vdash e 1 : \tau}{\lambda; \gamma \vdash e := e 1 : \tau \text{ cmd}}$
(ARRALLOC)	$\frac{\lambda(a) = \tau 1 \text{ arr } \tau 2, \gamma \vdash e 1 : \tau 2}{\lambda; \gamma \vdash \text{allocate } a[e 1] : \tau 2 \text{ cmd}}$
(ARRASSIGN)	$\frac{\lambda(a) = \tau 1 \text{ arr } \tau 2 \quad \lambda; \gamma \vdash e 1 : \tau 1 \quad \lambda; \gamma \vdash e 2 : \tau 1}{\lambda; \gamma \vdash a[e 1] := e 2 : \tau 1 \text{ cmd}}$
(SEQUENCE)	$\frac{\lambda; \gamma \vdash c : \tau \text{ cmd} \quad \lambda; \gamma \vdash c 1 : \tau \text{ cmd}}{\lambda; \gamma \vdash c; c 1 : \tau \text{ cmd}}$
(IF)	$\frac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma \vdash c : \tau \text{ cmd} \quad \lambda; \gamma \vdash c 1 : \tau \text{ cmd}}{\lambda; \gamma \vdash \text{if } e \text{ then } c \text{ else } c 1 : \tau \text{ cmd}}$
(WHILE)	$\frac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma \vdash c : \tau \text{ cmd}}{\lambda; \gamma \vdash \text{while } e \text{ do } c : \tau \text{ cmd}}$
(LETID)	$\frac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma[x : \tau \text{ var}] \vdash c : \tau 1 \text{ cmd}}{\lambda; \gamma \vdash \text{letid } x := e \text{ in } c : \tau 1 \text{ cmd}}$

Figure 2: Typing rules for secure information flow.

The rules for subtyping are shown in Figure 1. The typing rules are shown in Figure 2. A typing rule for only one arithmetic expression is shown, since rules for the other arithmetic expressions are defined in the same way. The typing rules correspond directly to certification conditions in the lattice model.

The typing rule **ARITH** is used to infer the types of arithmetic expressions. The rule says that if expressions e and $e1$ are of security type τ , then the type of the expression $e + e1$ is also τ . Note that if the types of e and $e1$ are different, it may be possible to make them the same by coercing one or both of them to higher security types, using the subtyping rules.

The **ASSIGN** rule checks the explicit information flow in assignment commands. The expressions e and $e1$ must have the same security type τ . If this is the case, the assignment command is given type τ *cmd*. If the types of e and $e1$ are not the same, and the type of $e1$ is lower than that of e , it may be possible to coerce the type of $e1$ to the type of e . However, if the type of $e1$ is higher than that of e , the assignment command causes information to flow from a high security type to a place of low security class, and the command is untypable.

The typing rules for **IF** and **WHILE** check whether the implicit flows are secure. These rules require that the guard expressions have the same security types as the commands in the branches or loop body, respectively, since there is an implicit flow from the guard to those commands. In addition, the two commands in the branches of **if**-statements must have the same type. As usual, coercion based on subtyping can help satisfy these constraints.

The **LETID** rule ensures that information flow in local variables declarations is secure. If a local variable x is initialized to the value of expression e of type τ , the identifier-type mapping γ is updated to map variable x to type τ while type checking the body of the **letid** command.

The **ARRACCESS** rule checks whether accesses to array elements are secure. If a is of type $\tau1$ *arr* $\tau2$ and expression e of type $\tau3$ indicates the index of the element of a being accessed, the array access expression has type $\tau1 \vee \tau3$, because there is information flow from both a and e to the result.

The **ARRALLOC** rule ensures that the information flow in array allocation is secure. If an array of type $\tau1$ *arr* $\tau2$ is being allocated, and the array's length is equal to the value of expression $e1$, then the type of $e1$ must be $\tau2$, because there is information flow from $e1$ to the array. If this is the case, the array allocation command is given the type $\tau2$ *cmd*, otherwise the command is untypable.

Given a program, and an environment of security types for locations accessed by the program, *type inference* is the process of inferring all possible types for each program node, if possible, so that the program is well-typed. Otherwise, type errors are reported. If the program is well-typed with respect to the secure information flow type system presented, information flow in the program is guaranteed to be secure.

3 Efficient Type Inference Algorithm and Data Structures

This section expresses type inference using extended Datalog rules and describes the generation of a specialized algorithm and data structures for type inference from the extended Datalog rules.

Type inference is generally done by using variables for unknown types of commands and expressions, and collecting constraints, in the form of type inequalities, that the type variables must satisfy for the program to be well-typed. These constraints characterize all typings of the program. The idea of our type inference algorithm is, given types for locations, to infer the lowest or highest security type for each program node, as appropriate, that the node can have in any typing of the program.

We define extended Datalog rules that we use to traverse the syntax tree of the program. The algorithm traverses the program top-down multiple times to infer minimum expression types, and then traverses the program bottom-up once to infer maximum command types.

3.1 Expressing type inference in extended Datalog rules

A Datalog program is a finite set of relational rules of the form

$$p_1(x_{11}, \dots, x_{1a_1}), \dots, p_h(x_{h1}, \dots, x_{ha_h}) \rightarrow q(x_1, \dots, x_a)$$

where h is a natural number, each p_i (respectively q) is a relation with a_i (respectively a) arguments, each x_{ij} and x_k is either a constant or a variable, and variables in x_k 's must be a subset of the variables in x_{ij} 's. If $h = 0$, then there are no p_i 's or x_{ij} 's, and x_k 's must be constants, in which case $q(x_1, \dots, x_a)$ is called a *fact*. For the rest of the paper, “rule” refers only to the case where $h \geq 1$, in which case each $p_i(x_{i1}, \dots, x_{ia_i})$ is called a *hypothesis* of the rule, and $q(x_1, \dots, x_a)$ is called the *conclusion* of the rule. The meaning of a set of Datalog rules and a set of facts is the smallest set of facts that contains all the given facts and all the facts that can be inferred, directly or indirectly, using the Datalog rules.

We use the following relations in our extended Datalog rules. We use two relations to map locations and arrays to their security types:

- `locenv(l, t)`: location `l` has type `t`.
- `arrenv(a, t1, t2)`: array `a` contains data of type `t1` and has length of type `t2`.

Together these two relations correspond to λ in the typing rules. The relations used to represent the syntax tree of the input program are:

- `root(c)`: `c` is the outermost command of a program.
- `literal(n)`: `n` is a literal.
- `loc(l)`: `l` is a location.
- `id(x)`: `x` is an identifier.
- `arith(e, e1, e2)`: expression `e` is an arithmetic expression with subexpressions `e1` and `e2` (e.g., `e` is `e1 + e2`).
- `assign(c, x, e)`: `c` is the command `x := e`.
- `if(c, e, c1, c2)`: `c` is the command `if e then c1 else c2`.
- `while(c, e, c1)`: `c` is the command `while e do c1`.
- `sequence(c, c1, c2)`: `c` is the command `c1; c2`.
- `letid(c, x, e, c1)`: `c` is the command `letid x := e in c1`.
- `arraccess(e, a, e1)`: `e` is the expression `a[e1]`.
- `arrassign(c, a, e1, e2)`: `c` is the command `a[e1] := e2`.
- `arrlen(e, a)`: `e` is the expression `a.length`.
- `arralloc(c, a, e1)`: `c` is the command `allocate a[e1]`.

The following relations are used to represent inferred types of program nodes and error messages about insecure information flow:

- **type**(p, t): program node p has type t . There may be multiple **type** facts for a program node. It is only necessary to keep the one with the highest type inferred so far.
- **h_{type}**(c, t): the maximum type of command c is t . The maximum type for a command is the highest type the command can have for the program to type correctly.
- **error**(c): the program is untypable because there may be insecure information flow in command c . A fact of the **error** relation is inferred when an assignment or array assignment statement assigns data to a location or an element of the array, and the data has a higher security type than the location or array. As discussed in Section 3.3. we can give more detailed error messages based on the derivation of each inferred **error**(c) fact, e.g., specifying the command that caused the error.

The functions **Join**(t_1, t_2) and **Meet**(t_1, t_2) return, respectively, the least upper bound and the greatest lower bound of two security types t_1 and t_2 . **Join** and **Meet** are defined for any two security types, since the types form a lattice. We can either precompute the least upper and greatest lower bound for each possible pair of security types, or compute them as needed during type inference, possibly with memoization. Efficient algorithms to compute **Meet** and **Join** are presented by Hassan et al. in [3]. The authors present three different algorithms for computing least upper bound and greatest lower bound: one is based on a transitive closure approach, the second is a more space-efficient method, and the last one employs a grouping technique based on modulation — it drastically reduces the code size, while keeping time complexity low. Time complexity of computing the complete least upper bound and greatest lower bound relations for a lattice is $O(s^2 \times \log s)$, where s is the size of the lattice. Time complexity for computing least upper bound or greatest lower bound for a single pair of types is $O(\log s)$.

The extended Datalog rules used for type inference are shown in Figures 3 and 4. The typing rules in Figure 2 can be written directly as extended Datalog rules, but efficient analysis needs to follow a predetermined procedure of traversing the program top-down multiple times to infer minimum expression types, and then traversing the program bottom-up once to infer maximum command types using the minimum types for expression. During the top-down traversals at any point in the evaluation we keep only the minimum inferred type for each expression. This is done for efficiency reasons, and it does not affect the correctness of the algorithm since only minimum types for expressions are used to infer maximum types for commands. We have rewritten the rules to embody this procedure. The rules in Figure 3 infer minimum types for expressions; the rules in Figure 4 infer maximum types for commands.

The rules are sound and complete with respect to the typing and subtyping rules in Section 2. Soundness is the property that if our rules infer a typing, expressed as the **type** relation for expressions and the **h_{type}** relation for commands, then types for expressions and commands in it satisfy the typing rules in Section 2. With the subtyping rules, higher expression types and lower command types also satisfy the rules. The soundness of our type inference algorithm can be proved by a structural induction.

Completeness is the property that if a typing satisfies the typing rules in Section 2, then our rules infer a typing too, and our inferred expression types, expressed in the **type** relation, are the lowest expression types that satisfy the typing rules in Section 2, and our command types, expressed as the **h_{type}** relation, are the highest command types that satisfy the typing rules in Section 2. The completeness of our type inference algorithm can be proved by an induction on derivations that use our rules.

```

(ROOT)
1.  root(c) → type(c, bottom)

(LITERAL)
2.  literal(n) → type(n, bottom)

(LOC)
3.  loc(l), locenv(l, t) → type(l, t)

(ARLEN)
4.  arrlen(e, a), arrenv(a, t1, t2) → type(e, t2)

(ARRACCESS)
5.  arraccess(e, a, e1), arrenv(a, t1, t2), type(e1, t3) → type(e, Join(t1, t3))

(ARITH)
6.  arith(e, e1, e2), type(e1, t1), type(e2, t2) → type(e, Join(t1, t2))

(ASSIGN ID)
7.  assign(c, x, e), id(x), type(e, t1), type(c, t2), type(x, t3) → type(x, Join(t1, t2, t3))

(ASSIGN LOC)
8.  assign(c, l, e), loc(l), type(l, t1), type(e, t2), not t2 ⊆ t1 → error(c)
9.  assign(c, l, e), loc(l), type(l, t1), type(c, t2), not t2 ⊆ t1 → error(c)

(ARRALLOC)
10. arralloc(c, a, e1), arrenv(a, t1, t2), type(e1, t3), not t3 ⊆ t2 → error(c)

(ARRASSIGN)
11. arrassign(c, a, e1, e2), arrenv(a, t1, t2), type(e1, t3), not t3 ⊆ t1 → error(c)
12. arrassign(c, a, e1, e2), arrenv(a, t1, t2), type(e2, t4), not t4 ⊆ t1 → error(c)

(SEQUENCE)
13. sequence(c, c1, c2), type(c, t) → type(c1, t)
14. sequence(c, c1, c2), type(c, t) → type(c2, t)

(IF)
15. if(c, e, c1, c2), type(e, t1), type(c, t2) → type(c1, Join(t1, t2))
16. if(c, e, c1, c2), type(e, t1), type(c, t2) → type(c2, Join(t1, t2))

(WHILE)
17. while(c, e, c1), type(e, t1), type(c, t2) → type(c1, Join(t1, t2))

(LETID)
18. letid(c, x, e, c1), type(e, t) → type(x, t)
19. letid(c, x, e, c1), type(c, t) → type(c1, t)

```

Figure 3: Extended Datalog rules for inference of minimum expression types and associated command types.

```

(ASSIGN ID MAX) 20. assign(c, x, e), id(x), type(x, t) → htype(c, t)
(ASSIGN LOC MAX) 21. assign(c, l, e), loc(l), type(l, t) → htype(c, t)
(ARRALLOC MAX) 22. arralloc(c, a, e1), arrenv(a, t1, t2) → htype(c, t2)
(ARRASSIGN MAX) 23. arrassign(c, a, e1, e2), arrenv(a, t1, t2) → htype(c, t1)
(SEQUENCE MAX) 24. sequence(c, c1, c2), htype(c1, t1), htype(c2, t2) → htype(c, Meet(t1, t2))
(IF MAX) 25. if(c, e, c1, c2), htype(c1, t1), htype(c2, t2) → htype(c, Meet(t1, t2))
(WHILE MAX) 26. while(c, e, c1), htype(c1, t) → htype(c, t)
(LETID MAX) 27. letid(c, x, e, c1), htype(c1, t) → htype(c, t)

```

Figure 4: Extended Datalog rules for inference of maximum command types.

3.2 Generation of efficient algorithm and data structures

We transform the extended Datalog rules into an efficient implementation using the method in [15] for Datalog rules. Two small extensions are needed, to handle negation, which in our rules simply requires a constant time check, and external functions, such as `Meet` and `Join`. The method has three steps.

- **Step 1:** transform the least fixed point (LFP) specification of the extended Datalog rules into a `while` loop.
- **Step 2:** transform expensive set operations in the loop into incremental operations.
- **Step 3:** design appropriate data structures for each set, so that operations on it can be implemented efficiently.

These three steps correspond to dominated convergence [7], finite differencing [18], and real-time simulation [17], respectively, as studied by Paige et al.

Auxiliary relations. For each rule with more than two hypotheses, we transform it to multiple rules with two hypotheses each. The transformation introduces auxiliary relations with necessary arguments to combine two hypotheses at a time. We repeatedly apply the following transformations to each rule with more than two hypotheses until only rules with at most two hypotheses are left. We replace any two hypotheses of the rule, say $P_i(X_{i1}, \dots, X_{ia_i})$ and $P_j(X_{j1}, \dots, X_{ja_j})$ by a new hypothesis, $Q(X_1, \dots, X_a)$, where Q is a fresh relation, and X_k 's are variables in the arguments of P_i or P_j that occur also in the arguments of other hypotheses or the conclusion of this rule. The name of the fresh relation is a concatenation of the names of the hypotheses of the new rule. We add a new rule: $P_i(X_{i1}, \dots, X_{ia_i}), P_j(X_{j1}, \dots, X_{ja_j}) \rightarrow Q(X_1, \dots, X_a)$.

For example, the `ARITH` rule is transformed into two rules with two hypotheses each as follows:

$$\begin{aligned} \text{arith}(e, e1, e2), \text{type}(e1, t1) &\rightarrow \text{arithType}(e, e1, e2, t1) \\ \text{arithType}(e, e1, e2, t1), \text{type}(e2, t2) &\rightarrow \text{type}(e, \text{Join}(t1, t2)) \end{aligned} \quad (1)$$

One auxiliary relation has been introduced — `arithType(e, e1, e2, t1)`, which denotes that `e` is an expression that performs an arithmetic operation on expressions `e1` and `e2`, and the type of `e1` is `t1`.

Fixed-point specification and while loop. We represent a relation of the form $Q(a_1, a_2, \dots, a_n)$ using tuples of the form $[Q \ a_1 \ a_2 \ \dots \ a_n]$. S with X and S less X denote $S \cup \{X\}$ and $S - \{X\}$, respectively. x in S and x not in S denote $x \in S$ and $x \notin S$, respectively. We use the notation $\{X : Y_1 \text{ in } S_1, \dots, Y_n \text{ in } S_n \mid Z\}$ for set comprehension. Each Y_i enumerates elements of S_i ; for each combination of values Y_1, \dots, Y_n , if the value of boolean expression Z is true, then the value of expression X forms an element of the resulting set. If Z is omitted, it is implicitly the constant `true`.

$\{[X_1 \ Y_1] \dots [X_n \ Y_n]\}$ denotes a map that maps X_1 to Y_1, \dots, X_n to Y_n . $\text{dom}(E)$ denotes the domain set of map E , i.e., $\{X : [X \ Y] \text{ in } E\}$. $E\{X\}$ denotes the *image set* of X under map E , i.e., $\{Y : [X \ Y] \text{ in } E\}$. $E\{X\} := S$ denotes setting the image set $E\{X\}$, of X under map E , to S . $\text{LFP}(S_0, F)$ denotes the smallest set S that satisfies the conditions $S_0 \subseteq S$ and $F(S) = S$.

The algorithm is expressed using standard control constructs `while`, `for`, `if`, and `case`. Program block structure is indicated by indentation. We abbreviate $X := X \text{ op } Y$ as $X \text{ op} := Y$.

The inputs to the algorithm are the given program represented by a set `program` of facts, and the relations `locenv` and `arrenv`, which map locations and arrays to security types. We let `input` be the set of facts in `program`, `locenv`, and `arrenv`, represented as tuples as described above.

```

input =
{[locenv l t]: locenv(l,t) in locenv} ∪
{[arrenv a t1 t2]: arrenv(a,t1,t2) in arrenv} ∪
{[root c]: root(c) in program} ∪
{[literal n]: literal(n) in program} ∪
{[loc l]: loc(l) in program} ∪
{[arrlen e a]: arrlen(e,a) in program} ∪
{[arraccess e a e1]: arraccess(e,a,e1) in program} ∪
{[arith e e1 e2]: arith(e,e1,e2) in program} ∪
{[assign c e1 e2]: assign(c,e1,e2) in program} ∪
{[arralloc c a e1]: arralloc(c,a,e1) in program} ∪
{[arrassign c a e1 e2]: arrassign(c,a,e1,e2) in program} ∪
{[sequence c c1 c2]: sequence(c,c1,c2) in program} ∪
{[if c e c1 c2]: if(c,e,c1,c2) in program} ∪
{[while c e c1]: while(c,e,c1) in program} ∪
{[letid c x e c1]: letid(c,x,e,c1) in program}

```

Given any set R of facts, and an extended Datalog rule with rule number n and with relation e in the conclusion, let $ne(R)$, be the set of all facts that can be inferred by that rule in one step given the facts in R . Here we use as an example the extended Datalog rules corresponding to the `sequence` commands. The sets $ne(R)$ for other rules are defined in the same way.

$$\begin{aligned}
13type(R) &= \{[type\ c1\ t]: \\
&\quad [sequence\ c\ c1\ c2] \text{ in } R, \\
&\quad [type\ c\ t] \text{ in } R\} \\
14type(R) &= \{[type\ c2\ t]: \\
&\quad [sequence\ c\ c1\ c2] \text{ in } R, \\
&\quad [type\ c\ t] \text{ in } R\}
\end{aligned} \tag{2}$$

For an auxiliary relation e introduced when splitting a rule with more than two hypotheses, let $e(R)$ be the set of facts that can be inferred by the rule defining e in one step given the facts in R .

The meaning of the given facts and the extended Datalog rules used for inferring minimum types for expressions is $LFP(\{\}, F)$, where $F(R)$ is the union of the input and the set of all facts that can be inferred by all rules for computing types and errors in one step given the facts in R , that is:

$$\begin{aligned}
LFP(\{\}, F), \text{ where } F(R) &= \text{input} \cup \\
&1type(R) \cup 2type(R) \cup 3type(R) \cup 4type(R) \cup \text{arraccessArrenv}(R) \cup 5type(R) \cup \\
&\text{arithType}(R) \cup 6type(R) \cup \text{assignID}(R) \cup \text{assignIdType}(R) \cup \text{assignIdTypeType}(R) \cup \\
&7type(R) \cup \text{assignLoc}(R) \cup \text{assignLocType}(R) \cup 8error(R) \cup 9error(R) \cup \\
&\text{arrallocArrenv}(R) \cup 10error(R) \cup \text{arrassignArrenv}(R) \cup 11error(R) \cup 12error(R) \cup \\
&13type(R) \cup 14type(R) \cup \text{ifType}(R) \cup 15type(R) \cup \text{whileType}(R) \cup 16type(R) \cup \\
&17type(R) \cup 18type(R) \cup 19type(R)
\end{aligned} \tag{3}$$

The set, O , of newly inferred facts about types is the difference between the above set and `input`, i.e.

$$O = LFP(\{\}, F) - \text{input}.$$

The resulting minimum types for expressions are the set, MIN , of maximum types, one for each expression, in O .

The meaning of the program facts, the resulting facts above, and the extended Datalog rules used for inferring maximum types for commands is $LFP(\{\}, F')$, where $F'(R)$ is the union of MIN , $input$, and the set of all facts that can be inferred by all rules for computing $htypes$ in one step given the facts in R , that is:

$$LFP(\{\}, F'), \text{ where } F'(R) = MIN \cup input \cup \\ assignId(R) \cup 20htype(R) \cup assignLoc(R) \cup 21htype(R) \cup 22htype(R) \cup \\ 23htype(R) \cup sequenceHtype(R) \cup 24htype(R) \cup ifHtype(R) \cup 25htype(R) \cup \\ 26htype(R) \cup 27htype(R) \quad (4)$$

The resulting set of maximum types for commands is:

$$O' = LFP(\{\}, F') - (MIN \cup input).$$

Efficient algorithms for computing O and O' are designed in the same way, so we show only the derivation of the algorithm for computing O . The least-fixed point expression $LFP(\{\}, F)$ is transformed into the following `while` loop:

```
R := {};
while exists x in F(R) - R:
  R with:= x;
```

The idea behind this transformation is to perform small update operations in each iteration of the `while` loop.

To maintain the result set O , which does not contain the input facts, i.e. $O = R - input$, we add to O every fact that is added to R except for facts that are in $input$. The above `while` loop, plus code to maintain O , is:

```
R := {};
O := {};
while exists x in F(R) - R:
  R with:= x;
  if x not in input:
    O with:= x; \quad (5)
```

Incremental computation. Next we transform expensive set operations in the loop into incremental operations. The idea is to replace each expensive expression exp in the loop with a variable, say E , and maintain the invariant $E = exp$, by inserting appropriate initializations and updates to E where variables in exp are initialized and updated, respectively.

The expensive expressions in type inference are all sets of facts inferred by each rule and a workset W . We use fresh variables to hold each of their respective values and maintain an invariant for each of these sets, in addition to one for the workset: $W = F(R) - R$. Here we show the invariants maintained for the sets in (2). The rest of the invariants are defined in the same way.

$$I13type = 13type(R) \\ I14type = 14type(R)$$

As an example of incremental maintenance of the value of an expensive expression, consider maintaining the invariant $I13type$. $I13type$ is the value of the set formed by joining elements from the `sequence` and `type` relations. $I13type$ can be initialized to $\{\}$ with the initialization $R = \{\}$. To update $I13type$ incrementally with the update $R \text{ with}:= x$, if x is of the form `[sequence c c1 c2]`

we consider all matching tuples of the form `[type c t]` and add each new tuple `[type c1 t]` to `I13type`. To form the tuples to be added, we need to efficiently find the appropriate values of variables that occur in `[type c t]` tuples, but not in `[sequence c c1 c2]`, i.e. the appropriate values of `t`, so we maintain an auxiliary map, called `type1_2`, that maps `c` to `t`. Symmetrically, if `x` is a tuple of the form `[type c t]`, we need to consider every matching tuple of the form `[sequence c c1 c2]` and add the corresponding tuple of the form `[type c1 t]` to `I13type`, so we need to efficiently find the value of variables that occur in `[sequence c c1 c2]` but not in `[type c t]`. Thus, we maintain an auxiliary map `sequence1_23` that maps `c` to `c1` and `c2`. These two auxiliary maps are shown below.

$$\begin{aligned} \text{type1_2} &= \{[[c] [t]] : [\text{type } c \ t] \text{ in } R\} \\ \text{sequence1_23} &= \{[[c] [c1 \ c2]] : [\text{sequence } c \ c1 \ c2] \text{ in } R\} \end{aligned}$$

Since $R = 0 \cup \text{input}$, the above two sets are equivalent to:

$$\begin{aligned} \text{type1_2} &= \{[[c] [t]] : [\text{type } c \ t] \text{ in } 0\} \\ \text{sequence1_23} &= \{[[c] [c1 \ c2]] : [\text{sequence } c \ c1 \ c2] \text{ in } \text{input}\} \end{aligned}$$

The first set of components in an auxiliary map is referred to as the *anchor* and the second set of elements as the *nonanchor*.

Thus, the algorithm can directly find only matching tuples and consider only combinations of facts that make both hypotheses true simultaneously, and it considers each combination only once. Similar auxiliary maps are maintained for all maintained invariants that are formed by joining elements from two relations.

All variables holding the values of expensive computations listed above and all auxiliary maps are initialized together with the assignment $R := \{\}$ and updated incrementally together with the assignment $R \text{ with}:= x$ in each iteration. We show the update for the addition of a fact of relation `sequence` only for `I13type` invariant and `sequence1_23` auxiliary map. Other updates are processed in the same way.

```
case x of [sequence c c1 c2]:
  I13type  $\cup:=$  {[type c1 t] : [t] in type1_2{[c]}};
  W  $\cup:=$  {[type c1 t] : [t] in type1_2{[c]} | [type c1 t] not in 0};
  sequence1_23  $\cup:=$  {[[c] [c1 c2]]};
```

(6)

Adding these initializations and updates, and other similar ones for the other cases, and replacing $F(R) - R$ with W in (5), we obtain the following complete code:

```
initialization;
R:={};
O:={};
while exists x in W:
  update using (6) and other similar updates for the other cases
  W less:= x;
  R with:= x;
  if x not in input:
    O with:= x;
```

We next eliminate dead code. To compute the result set `0`, only `input`, `W`, and the auxiliary maps are needed; `R` is dead because all uses of it are replaced with `0` and `input`; the sets for maintaining other invariants, such as `I13type` and `I14type`, are dead because $F(R) - R$ in the `while` loop was

replaced with W . We eliminate them from the initialization and updates. For example, eliminating them from the updates in (6), we get:

```

case x of [sequence c c1 c2]:
  W ∪:= {[type c1 t] : [t] in type1_2{[c]} | [type c1 t] not in 0};      (7)
  sequence1_23 ∪:= {[[c] [c1 c2]]};

```

The complete pseudocode for inferring minimum types of expressions and maximum types for commands is in the Appendix.

We clean up the code to contain only uniform operations and set elements. This simplifies data structure design. We decompose 0 and W into several sets, each corresponding to one relation in the extended Datalog rules. For example, 0 is decomposed into 0type and 0error , where 0type contains tuples of the `type` relation in 0 and 0error contains tuples of the `error` relation in 0 . This decomposition lets us eliminate relation names from the first component of tuples, with appropriate changes to the `while` clause and `case` clauses. Then, we apply the following three sets of transformations.

- (i) Transform operations on sets into loops that use operations on set elements. Each addition of a set is transformed to a `for` loop that adds the elements one at a time. For example, line 2 of (7) is transformed into:

```

for [t] in type1_2{[c]}:
  if [c1 t] not in 0type:
    Wtype with:= [c1 t];

```

- (ii) Replace tuples and tuple operations with maps and map operations. For example, the above `for` loop is transformed into:

```

for [c] in dom(type1_2):
  for [t] in type1_2{[c]}:
    if [c1 t] not in 0type:
      Wtype with:= [c1 t];

```

Transform `while` loops similarly. Also, for each membership in a map test, we replace $[X Y]$ `not in` M with Y `not in` $M\{X\}$. For example, the membership test `[c1 t] not in Rtype` is replaced with `t not in Rtype{c1}`.

Each addition to a map M `with:= [X Y]` is replaced with $M\{X\}$ `with:= Y`. For example, the addition to the workset `Wtype`.

```

Wtype with:= [c1 t];

```

is replaced with

```

Wtype{c1} with:= t;.

```

- (iii) Test for membership before adding or deleting an element of a set. Specifically, we replace each statement S `with:= X` with `if X not in S: S with:= X`.

Note that when removing an element from a workset, the membership test is unnecessary, since the element is retrieved from the workset. Also, when adding an element to a result set, the membership test is unnecessary, since elements are moved from the corresponding workset to the result set one at a time, and each element is put in the workset and thus in the result set only once.

Data structures. After the above transformations, each firing of an extended Datalog rule involves a constant number of set operations. Since each set operation takes worst-case constant time in the generated code, as described below, each firing takes worst-case constant time. Next we describe how to guarantee that each set operation takes worst-case constant time. The operations are of the following kinds: set initialization $S := \{\}$, computing image set $M\{X\}$, element retrieval **for** X **in** S and **while exists** X **in** S , membership test X **in** S and X **not in** S , element addition S **with** X , and element deletion S **less** X . Membership test and computing image set are called *associative access*.

A uniform method is used to represent all sets and maps, using arrays for sets that have associative access, linked lists for sets that are traversed by loops and both arrays and linked lists for sets that have both operations.

- **result sets:** Resultsets, such as `Otype`, are represented by nested array structures. A result set containing tuples with `a` components is represented using an `a`-level nested array structure. The first level is an array indexed by values in the domain of the first component of the result set; the `k`-th element of the array is null if there is no tuple in the result set whose first component has value `k`, and otherwise is `true` if `a=1`, and otherwise is recursively an `(a-1)`-level nested array structure for the remaining components of tuples in the result set whose first component has value `k`.
- **worksets:** Worksets corresponding to relations that occur in the conclusions of rules, such as `Wtype`, are represented by arrays and linked lists. Each workset is represented the same way as the corresponding result set with two additions. First, for each array we add a linked list containing indices of non-null elements of the array. Second, to each linked list we add a tail pointer, i.e., a pointer to the last element, so the list can be used as a queue. One or more records are used to put each array, linked list, and tail pointer together. Each workset corresponding to a relation that does not occur in the conclusion of any rule, is represented simply as a nested queue structure (without the underlying arrays), one level for each component of the tuples, linking the elements (instead of array indices) directly.
- **auxiliary maps:** Auxiliary maps, such as `sequence1.23`, are implemented as follows. Each auxiliary map for a relation that appears in an extended Datalog rule's conclusion uses a nested array structure for all components of the tuples and additionally linked lists for each non-anchor component. Each auxiliary map for a relation that does not appear in the conclusion of any rule uses a nested array structure for the anchor components, and nested linked-lists for the non-anchor components.

3.3 Informative Error Reporting

Informative error messages are essential for determining the sources of information flow errors and for fixing these errors and developing secure programs.

We can easily give meaningful error messages by adding rules that keep additional information during type inference. We implement the concepts needed to produce informative error messages, as

$$\begin{aligned}
& \text{id}(x) \rightarrow \text{prinVar}(x, x) \\
& \text{loc}(l) \rightarrow \text{prinVar}(l, l) \\
& \text{arrlen}(e, a) \rightarrow \text{prinVar}(e, a) \\
& \text{arraccess}(e, a, e1) \rightarrow \text{prinVar}(e, a) \\
& \text{arraccess}(e, a, e1), \text{prinVar}(e1, v) \rightarrow \text{prinVar}(e, v) \\
& \text{arith}(e, e1, e2), \text{prinVar}(e1, v) \rightarrow \text{prinVar}(e, v) \\
& \text{arith}(e, e1, e2), \text{prinVar}(e2, v) \rightarrow \text{prinVar}(e, v)
\end{aligned}$$

Figure 5: Rules for principal variables of expressions.

described by Deng and Smith [9], in extended Datalog rules, and apply the method presented above to generate efficient algorithms and data structures directly from the rules.

Following [9], we use two notions to collect information relevant to information flow errors — the *principal variables* for each expression, and the *security level history* for each variable. The *principal variables* for an expression are a minimum set of variables in the expression that can be used to determine the type of the expression. Intuitively, the principal variables for an expression can provide an explanation of the type of the expression. The *security level history* of a variable keeps track of the different security levels a variable had during type inference and the principal variables of the expression "responsible" for each change to the variable's security type. This history can be helpful for understanding how type errors occurred.

When a command generates a type error, the algorithm can provide the principal variables of the expressions that are part of the command, as well as the security level histories of all variables that are part of these expressions.

We use $\text{prinVar}(e, v)$ to denote that v is a principal variable for expression e . The rules for computing principal variables are shown in Figure 5.

We use $\text{hist}(v1, t, v2)$ to denote that, during type inference, the security type of variable $v1$ changed to t , and $v2$ is a principal variable of the expression causing the change. Since the security level of a variable can change only when an assignment to an identifier occurs, we need only one rule to keep the security level history:

$$\begin{aligned}
& \text{assign}(c, x, e), \text{id}(x), \text{type}(e, t1), \text{type}(c, t2), \text{type}(x, t3), \\
& \quad t3 \subset \text{Join}(t1, t2, t3), \text{prinVar}(e, v) \\
& \quad \rightarrow \text{hist}(x, \text{Join}(t1, t2, t3), v)
\end{aligned}$$

Error reports are based on the relation $\text{errReport}(c, v1, t, v2)$, which means that there may be insecure information flow in command c , $v1$ is a principal variable of e or a variable that transitively caused the security level of a principal variable to change, t is a new security level of $v1$ for a change recorded in $v1$'s security level history, and $v2$ is a variable that directly caused the security level of $v1$ to change. The rules defining the errReport relation are:

$$\begin{aligned}
& \text{error}(c), \text{assign}(c, l, e), \text{prinVar}(e, v1), \text{hist}(v1, t, v2) \rightarrow \text{errReport}(c, v1, t, v2) \\
& \text{error}(c), \text{arralloc}(c, a, e), \text{prinVar}(e, v1), \text{hist}(v1, t, v2) \rightarrow \text{errReport}(c, v1, t, v2) \\
& \text{error}(c), \text{arrassign}(c, a, e1, e2), \text{prinVar}(e1, v1), \text{hist}(v1, t, v2) \rightarrow \text{errReport}(c, v1, t, v2) \\
& \text{error}(c), \text{arrassign}(c, a, e1, e2), \text{prinVar}(e2, v1), \text{hist}(v1, t, v2) \rightarrow \text{errReport}(c, v1, t, v2) \\
& \text{errReport}(c, v1, t1, v2), \text{hist}(v2, t2, v3) \rightarrow \text{errReport}(c, v2, t2, v3)
\end{aligned}$$

The error report consists of all inferred facts of the `errReport` relation in order, together with the inferred types for variables mentioned in `errReport` facts.

4 Complexity Analysis

This section presents time and space complexity analysis for our type inference algorithm. We first analyze the complexity of the algorithm in Section 3.2 and then analyze the additional cost of the error reporting described in Section 3.3.

4.1 Time complexity

We analyze the time complexity of type inference by carefully bounding the number of facts actually used by the extended Datalog rules. For each rule we determine precisely the number of facts processed by it, avoiding where possible approximations that use the product of the sizes of individual argument domains.

Size parameters. We first define the size parameters used in the complexity analysis. The number of facts of a relation r that are given or can be inferred is called r 's *size*. The number of nodes in the input program is called the *program size* and is denoted by p . For a relation named r , $\#r$ denotes the size of r . We also use the following additional size parameters:

- **#array**: number of arrays in the program
- **#cmd**: number of commands in the program
- **#expr**: number of expressions in the program
- **p**: the size of the program, i.e. the number of program nodes
- **s**: the size of the lattice of security types
- **h**: the height of the lattice of security types

Analysis of time complexity. The time complexity for a set of Datalog rules is the total number of combinations of hypotheses considered in evaluating the rules. For each rule r , the number of firings for the rule is a count of: (i) for rules with one hypothesis: the number of facts which make the hypothesis true; (ii) for rules with two hypotheses: the number of combinations of facts that make the two hypotheses simultaneously true. The total time complexity is time for reading the input, plus the time for applying each logic rule.

It is possible to precompute all values for the functions `Join` and `Meet` in $O(s^2 \times \log s)$ time, and, if we do so, any of them can be looked up on $O(1)$ time. However, this may be unnecessary, since it is possible that not all values of these functions are needed. Therefore, we compute the values of `Join` and `Meet` as needed and memoize already computed values which can be looked up in $O(1)$ time if needed again. The time complexity of computing `Join` or `Meet` for two security types is $O(\log s)$. The type complexity of computing whether the subtyping relation holds between two types is $O(h)$.

The algorithm traverses the program top-down multiple times to infer the `type` relation, i.e. minimum expression types, and then traverses the program bottom-up once to infer the `htype`, i.e. maximum command types. Facts of the `type` relation for variables can be inferred by use of the `LETID` rules. New types for variables can be inferred by use of the `ASSIGN ID` rule. This can cause facts of

the `type` relation for other variables to be inferred. At any point in the evaluation at most one fact of the `type` relation is kept for a program node, and that is the one with the highest type for the program node that has been inferred so far. The type of each program node can be raised at most `h` times. Thus worst case time complexity for each of the extended Datalog rules for type inference is equal the program size multiplied by the height of the lattice of security types and the time to compute `Join` and `Meet`, i.e. $O(p \times h \times \log s)$.

The additional algorithms for inferring principal variables and keeping security level history of variables have time complexity $O(p \times pVars \times s)$, where `pVars` is the maximum number of principal variables for an expression, and `s` is the size of the lattice of security types. The algorithm for error reporting has worst-case time complexity $O(\#error \times pVars^2 \times s)$, where `#error` is the number of information flow errors inferred and $\#error \leq (\#assign + \#arralloc + \#arassign)$.

4.2 Space complexity

To analyze space complexity we consider the space needed beyond the space taken by the given program, lattice, and location and array type assignment relations `locenv` and `arrenv`. The total such space is the sum of the space needed for the result sets `Otype`, `Oerror`, and `O'htype`, the worksets `Wtype`, `Werror`, and `Whtype`, and the space needed for all auxiliary maps. Worksets take the same space asymptotically as the result sets, so we will not consider them separately here. We refer to the space taken by the result sets `Otype`, `Oerror`, and `O'htype` as *output space*. We refer to the space taken by all auxiliary maps as *auxiliary space*. Here we show the space complexity computation for the `ARITH` rule. Space complexity is computed in the same way for all remaining rules.

Analysis of output space. A result set is created for each relation that occurs in the conclusion of a rule, i.e., each relation for which new facts may be inferred, namely, `type`, `htype`, and `error`. The space taken by the result set for a relation is clearly bounded by the product of the sizes of its arguments' domains.

For the `type` relation, the argument domains are program constructs and types in the security type lattice, so the output space for it is $O(p \times s)$. The argument domains of the `htype` relation are commands in the program and types in the security type lattice; the output space for it is $O(\#cmd \times s)$. The argument of the `error` relation is a command, so the output space for this relation is $O(\#cmd)$, which is bounded above by $O(p)$. We conclude that the total asymptotic output space complexity is $O(p \times s)$.

To analyze the output space for informative error reporting, we consider the relations `prinVar`, `hist`, and `errReport`. The asymptotic output space for error reporting is dominated by the space for the `errReport` relation, which has four arguments - a command, two variables, and a type. The output space is $O(\#cmd \times \#id^2 \times s)$.

Analysis of auxiliary space. Auxiliary maps are created only for rules that have two hypotheses. The space needed by an auxiliary map depends on the operations that need to be performed on it. We distinguish between two kinds of auxiliary maps — ones that need to support the image set operation only, and ones that need to support the image operation and membership test. For maps in the first category, space is needed for the nested arrays for the anchors and the nested linked lists for the non-anchors. For maps in the second category, space is needed for the nested arrays for all components and the nested linked lists for the nonanchors, however the latter does not affect the asymptotic space needed.

Here we show the auxiliary space computation for the rules (1) for arithmetic expressions. The others are very similar. Four auxiliary maps are needed for these rules: in terms of the arguments of these rules, `arith2_13` maps `[e1]` to `[e e2]`, `type1_2` maps `[e1]` to `[t1]`, and `arithType3_124` maps `[e2]` to `[e e1 t1]`. The size of `type1_2` is bounded by the size of the `type` relation, which is

auxiliary map	[anchor nonanchor]	rules that need it	space
arrlen2.1	[[a] [e]]	4	$O(\#expr + \#array)$
arraccess2.13	[[e] [a e1]]	5	$O(\#expr + \#array)$
arraccessArrenv2.12	[[a] [t1 t2]]	5	$O(\#array + s)$
arith2.13	[[e1] [e e2]]	6	$O(\#expr)$
type1.2	[[e1] [t1]]	5-21	$O(p \times s)$
arithType2.13	[[e2] [e t1]]	4	$O(\#expr + s)$
assign2.13	[[x] [c e]]	7,8,9,20,21	$O(p)$
assignId2.13	[[e] [c x]]	7,20	$O(p)$
assignIdType3.124	[[x] [c t1]]	7	$O(p + s)$
assignIdTypeType2.134	[[x] [c t1 t2]]	7	$O(p + s)$
assignLoc2.13	[[e] [c x]]	8,9,21	$O(p)$
assignLocType1.234	[[c] [x e t1]]	9	$O(p + s)$
assignLocType3.124	[[l] [c e t1]]	8	$O(p + s)$
arralloc2.13	[[a] [c e1]]	10,22	$O(p)$
arrenv1.23	[[a] [t1 t2]]	10,11,12,22,23	$O(\#array \times s^2)$
arrallocArrenv4.123	[[a] [c e1 t1]]	10,22	$O(p)$
arrassign2.134	[[a] [c e1 e2]]	11,12,23	$O(p)$
arrassignArrenv2.134	[[e1] [c a e2 t1]]	11,12	$O(p+s)$
sequence1.23	[[c] [c1 c2]]	13,14,24	$O(\#cmd)$
if2.134	[[e] [c c1 c2]]	15,16,25	$O(p)$
ifType1.234	[[c] [c1 c2 t1]]	15,16	$O(p + s)$
while2.13	[[e] [c c1]]	17,26	$O(\#cmd + \#expr)$
whileType1.234	[[c] [e c1 t1]]	17	$O(\#expr + \#cmd + s)$
letid3.124	[[e] [c x c1]]	18	$O(\#expr + \#cmd)$
letid1.234	[[c] [x e c1]]	19	$O(p)$
letid4.123	[[c1] [c x e]]	27	$O(p)$
hType1.2	[[c1] [t1]]	24,25,26,27	$O(p \times s)$
sequenceHType2.13	[[c2] [c c1 t1]]	24	$O(\#cmd + s)$
if3.124	[[c] [e c1 c2]]	25	$O(\#cmd + \#expr)$
ifHType3.124	[[c2] [c e c1 t1]]	25	$O(\#cmd + \#expr + s)$
while1.23	[[c] [e c1]]	26	$O(\#cmd + \#expr)$
total auxiliary space			$O(\#array \times s^2 + p \times s)$

Figure 6: Auxiliary space used for type inference.

$O(p \times s)$, as discussed above. The other two auxiliary maps need to support the image set operation only. The space required for `arith2.13` is the size of the array for the anchor plus the sum of the sizes of all the linked lists for the non-anchor components. The former is $O(\#expr)$. The latter is bounded by the size of the `arith` relation. So, the space needed for `arith2.13` is $O(\#expr + \#arith)$, which is $O(\#expr)$. The space need for `arithType3.124` is calculated in the same way and is also $O(\#expr)$. Thus the total space for these auxiliary maps is $O(p \times s)$.

Auxiliary space for type inference for information flow is summarized in Figure 6. The first column gives the names of all auxiliary maps used. The second column shows their anchors and nonanchors. The third column lists the rules that share each map. The fourth column shows the space needed for the map.

The total auxiliary space needed is $O(\#array \times s^2 + p \times s)$. The total space complexity of type inference for secure information flow is thus $O(\#array \times s^2 + p \times s)$.

Description of Program	Number of Program Nodes	Time to Infer Expression Types		Time to Infer Command Types(ms)
		Total (ms)	Per node (μ s)	
Thermostat	89	6	67	10
ralphSIS	150	10	77	14
Safety Injection System	159	12	75	16
Shutdown Control Logic for a Nuclear Power Plant	411	32	78	44
Cruise Control System	465	36	77	50
FDIR	519	38	73	58
Contol Panel	3471	370	107	904

Table 1: Time to infer minimum types for expressions and maximum types for commands.

5 Experimental Results

To experimentally confirm our time complexity calculations, we generated an implementation of our algorithm in Python. The generated implementation consists of 900 lines of Python code. We analyzed programs of varying size, to determine how the running time of the algorithm scales with program size. For each program, we report the CPU time for the analysis, using Python 2.3.5 on a 500MHz Sun Blade 100 with 256 Megabytes of RAM, running SunOS 5.8. Reported times are averaged over 10 trials. We use two security types in these experiments, *low* and *high*. All timing data shown is for experiments with all global variables having the *low* security type.

Since the type system supports a relatively small number of operations, finding programs for real applications it could analyze proved a challenge. We overcame this by analyzing programs generated from SCR specifications [14], including specifications for real applications. SCR specifications use a tabular notation, built on top of a state machine model, to specify the behavior of a system. We modified OSCAR, a code generator for SCR [21], to generate programs using only the operations the type system supports. This involved adding an outer loop that waits for events, rather than using function calls to notify the generated code of events. We also extended OSCAR to output the abstract syntax tree of the generated code as Datalog facts. This allows us to analyze realistic systems.

Table 1 gives the results for inferring minimum types of expressions and maximum types of commands. The second column gives the program size, expressed as the number of nodes in the abstract syntax tree. The third column gives the CPU time required to infer minimum expression types for each program. The fourth column gives the time per fact required to infer minimum types of expressions, which should remain constant. Indeed, it is nearly the same for all programs except the smallest and largest; we suppose the variation is due to the memory hierarchy. The final column gives the CPU time taken to infer the maximum types of commands. The results show that CPU for inferring minimum time for expressions is linear in the number of program nodes. The maximum types of commands were inferred given the inferred minimum types of expressions. Since the input is the set of types of all program nodes, but maximum types are inferred just for commands, the time complexity was linear in a combination of the number of program nodes and the number of commands in the program.

6 Related Work and Conclusions

A large amount of research has been done on information flow analysis since Denning’s pioneering work [10, 11]. A survey of language-based information flow security appears in [22]. Various analysis frameworks have been used, including abstract interpretation, e.g., [6, 4, 12, 13], and type systems, e.g., [27, 28, 16, 20, 23, 26, 9].

Type-based approaches have been studied extensively, because types are inherently compositional, provide good documentation as well as correctness guarantees, and seem more familiar to programmers (who are familiar with standard type systems). As the survey [22] shows, there are many information-flow type systems. We focus here on the first type systems, where type inference was open for a decade, and the eventual development of such an algorithm only yielded a quadratic algorithm. The difficulty of type inference depends on many factors, notably what language constructs are used, and whether the security levels, which in general form a partial order, are assumed to form a lattice.

The first information-flow type system [27], by Volpano, Irvine and Smith, is for a simple imperative programming language with local variables. A decade later, Deng and Smith give a type inference algorithm for this language extended with arrays but without local variables and assuming the security levels form a lattice [9]. Their algorithm uses explicit iteration to compute a least fixed point. The worst-case time complexity of their algorithm is quadratic in the size of the program and linear in the height of the lattice. Their time complexity analysis assumes that joins can be computed in constant time. They do not give the space complexity.

We give an algorithm for a type system with both local variables and arrays, also assuming that security levels form a lattice. Our algorithm is linear in the size of the program. We also give the precise space complexity, which is also linear in the size of the program. At a high level, their algorithm and our algorithm are very similar. The main difference is that, by expressing the algorithm using rules and applying a systematic implementation method, we obtain an implementation that performs much more efficiently by incremental computation.

An obvious open problem is to develop an efficient information flow type inference algorithm for languages with additional features. Volpano and Smith give a type inference algorithm for the language in [27] extended with polymorphic procedures [28]. Recent work on type-based information-flow security considers many additional features found in modern programming languages, such as dynamically allocated mutable objects, subclassing, method overriding, type casts, dynamic type tests, and exceptions [16, 20, 23, 26].

In short, while several information-flow analysis algorithms exist, they have been developed manually under different assumptions and for different language features and different definitions of information flow, so it is difficult to compare them. Furthermore, relatively little is known about the worst-case or typical time complexity of these algorithms. This leaves many problems for future research.

In summary, this paper presents an approach to systematically deriving efficient algorithms for type inference for secure information flow types. We applied the approach to a classic information flow type system [27, 9] and obtained an efficient type inference algorithm and a precise characterization of its time complexity. We also implemented the algorithm and performed experiments on code generated from high-level specifications of real systems. We plan to apply the approach to information flow type systems for richer programming languages and compare the efficiency and precision of the resulting algorithms.

References

- [1] M. Abadi. Secrecy by typing in cryptographic protocols. In *Proceedings of Theoretical Aspects of Computer Software (TACS'97)*, volume 1281 of *Lecture Notes in Computer Science*, pages 611–638, Berlin, Germany, 1997. Springer.
- [2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [3] H. Ait-Kaci, R. S. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *Programming Languages and Systems*, 11(1):115–146, 1989.

- [4] M. Avvenuti, C. Bernardeschi, and N. D. Francesco. Java bytecode verification for secure information flow. *SIGPLAN Notices*, 38(12):20–27, 2003.
- [5] J.-P. Banâtre, C. Bryce, and D. L. Métayer. Compile-time detection of information flow in sequential programs. In *ESORICS '94: Proceedings of the Third European Symposium on Research in Computer Security*, pages 55–73, London, UK, 1994. Springer-Verlag.
- [6] R. Barbuti, C. Bernardeschi, and N. D. Francesco. Checking security of Java bytecode by abstract interpretation. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 229–236, New York, NY, USA, 2002. ACM Press.
- [7] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, 1989.
- [8] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. 1990. Springer-Verlag.
- [9] Z. Deng and G. Smith. Type inference and informative error reporting for secure information flow. In *Proceedings of ACMSE 2006: 44th ACM Southeast Conference*, Melbourne, Florida, 2006.
- [10] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
- [11] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
- [12] N. D. Francesco, A. Santone, and L. Tesei. Abstract interpretation and model checking for checking secure information flow in concurrent systems. *Fundam. Inf.*, 54(2-3):195–211, 2003.
- [13] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 186–197, New York, NY, USA, 2004. ACM Press.
- [14] C. Heitmeyer. Using the SCR* toolset to specify software requirements. In *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques (WIFT)*, page 12, Washington, DC, USA, 1998. IEEE Computer Society.
- [15] Y. A. Liu and S. D. Stoller. From Datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 172–183. ACM Press, 2003.
- [16] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, Texas, Jan. 1999. ACM Press.
- [17] R. Paige. Real-time simulation of a set machine on a RAM. In *Proceedings of the International Conference on Computing and Information*, volume 2, pages 68–73, 1989.
- [18] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):402–454, 1982.
- [19] J. Palsberg and P. Ørbæk. Trust in the lambda-calculus. In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 314–329, London, UK, 1995. Springer-Verlag.

- [20] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 319–330, New York, NY, USA, 2002. ACM Press.
- [21] T. Rothamel, C. Heitmeyer, E. Leonard, and Y. A. Liu. Generating optimized code from SCR specifications. *Proceedings of LCTES 2006: ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2006.
- [22] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal On Selected Areas in Communications*, 21(1):5–19, January 2003.
- [23] V. Simonet. Flow CAML in a nutshell. In G. Hutton, editor, *Proceedings of the First APPSEM-II Workshop*, pages 152–165, 2003.
- [24] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, New York, NY, USA, 1998. ACM Press.
- [25] M. Sulzmann. A general type inference framework for Hindley/Milner style systems. In *FLOPS '01: Proceedings of the 5th International Symposium on Functional and Logic Programming*, pages 248–263, London, UK, 2001. Springer-Verlag.
- [26] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Proceedings of the 11th International Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 84–99, Aug. 2004.
- [27] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.
- [28] D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, London, UK, 1997. Springer-Verlag.

APPENDIX

Pseudocode for inferring minimum types of expressions:

```
W := input;
locenv1_2={}; arrlen2_1 := {}; arrenv1_23 := {}; type1_2 := {};
arraccess2_13 := {}; arraccessArrenv2_13 := {};
arith2_13 := {}; arithType2_13 := {};
assign2_13 := {}; assignIdType1_23 := {}; assignIdTypeType2_134 := {};
assignLoc2_13 := {}; assignLocType3_124 := {}; assignLocType1_234 := {};
arralloc2_13 := {}; arrallocArrenv3_124 := {};
arrassign2_134 := {}; arrassignArrenv2_134 := {}; arrassignArrenv1_234 := {};
sequence1_23 := {}; if2_134 := {}; ifType1_234 := {};
while2_13 := {}; whileType1_23 := {}; letid3_124 := {}; letid1_234 := {};
0 := {};

while exists x in W:

  case x of [root x]:
    //update for rule 1
    if [type x bottom] not in 0:
      W with:= [type x bottom];

  case x of [literal n]:
    //update for rule 2
    if [type n bottom] not in 0:
      W with:= [type n bottom];

  case x of [loc l]:
    //update for rule 3
    W U:= {[type l t] : [t] in locenv1_2{[l]} | [type l t] not in 0};
    //update for rules 8 and 9
    W U:= {[assignLoc c l e] : [c e] in assign2_13{[l]} | [assignLoc c l e] not in 0};

  case x of [locenv l t]:
    //update for rule 3
    if [loc l] in W and [type l t] not in 0:
      W with:= [type l t];
      locenv1_2 with:=[l],[t];

  case x of [arrlen e a]:
    //updates for rule 4
    W U:= {[type e t2] : [t1 t2] in arrenv1_23{[a]} | [type e t2] not in 0};
    //auxiliary map used for rule 4
    arrlen2_1 with:= [[a], [e]];

  case x of [arrenv a t1 t2]:
    //update for rule 4
    W U:= {[type e t2] : [e] in arrlen2_1{[a]} | [type e t2] not in 0};
    //update for rule 5
    W U:= {[arraccessArrenv e e1 t1] : [e e1] in arraccess2_13{[a]} |
      [arraccessArrenv e e1 t1] not in 0};

    //update for rule 10
    W U:= {[arrallocArrenv c a e1 t1] : [c e1] in arralloc2_13{[a]} |
      [arrallocArrenv c a e1 t1] not in 0};

    //update for rules 11 and 12
    W U:= {[arrassignArrenv c a e1 t1] : [c e1 e2] in arrassign2_134{[a]} |
      [arrassignArrenv c a e1 t1] not in 0};
    //update to auxiliary map used for rules 10, 11 and 12
```

```

arrenv1_23 with:= [[a], [t1 t2]];

case x of [arith e e1 e2]:
  //update for rule 6
  W U:= {[arithType e e2 t1] : [t1] in type1_2{[e1]} | [arithType e e2 t1] not in 0};
  //update to auxiliary map used for rule 6
  arith2_13 with:= [[e1], [e e2]];

case x of [arraccess e a e1]:
  //update for rule 5
  W U:= {[arraccessArrenv e e1 t1] : [t1 t2] in arrenv1_23{[a]} |
      [arraccessArrenv e e1 t1] not in 0};
  //update to auxiliary map used for rule 5
  arraccess2_13 with:= [[a], [e e1]];

//arraccessArrenv is an auxiliary relation for the first two hypotheses of rule 5
case x of [arraccessArrenv e e1 t1]:
  //update for rule 5
  W U:= {[type e t1] : [t3] in type1_2{[e1]} | [type e t1] not in 0};
  //update to auxiliary map mapping e1 to e and t1 where arraccessArrenv(e,e1,t1)
  arraccessArrenv2_13 with:= [[e1], [e t1]];

//arithType is an auxiliary relation for the first two hypotheses of rule 6
case x of [arithType e e2 t1]:
  //update for rule 6
  W U:= {[type e t1] : [t2] in type1_2{[e2]} | [type e Join(t1,t3)] not in 0};
  //update to auxiliary map used for rule 6
  arithType2_13 with:= [[e2], [e t1]];

case x of [assign c var e]:
  //update for rule 7
  W U:= {[assignId c var e] | [assignId c var e] not in 0};
  //update for rules 8 and 9
  W U:= {[assignLoc c var e] | [assignLoc c var e] not in 0};
  assign2_13 with:= [[var], [c e]];

case x of [Id x]:
  //update for rule 7
  W U:= {[assignId c x e] : [c e] in assign2_13{[x]} | [assignId c x e] not in 0};

//assignId is an auxiliary relation for the first two hypotheses of rule 7
case x of [assignId c x e]:
  //update for rule 7
  W U:= {[assignIdType c x t1] : [t1] in type1_2{[e]} | [assignIdType c x t1] not in 0};
  //auxiliary map - maps e to c and x where assignId(c,x,e)
  assignId3_12 with:= [[e], [c x]];

//assignIdType is an auxiliary relation for assignId and the third hypothesis of rule 7
case x of [assignIdType c x t1]:
  //update for rule 7
  W U:= {[assignIdTypeType c x t1 t2] : [t2] in type1_2{[c]} | [assignIdTypeType c x t1 t2] not in 0};
  //auxiliary map used for rule 7
  assignIdType1_23 with:= [[c], [x t1]];

//assignIdTypeType is an auxiliary relation for assignIdType and the fourth hypothesis of rule 7
case x of [assignIdTypeType c x t1 t2]:
  //update for rule 7
  W U:= {[type x t1] : [t3] in type1_2{[x]} | [type x Join(t1,t2,t3)] not in 0};
  //auxiliary map used for rule 7

```

```

assignIdTypeType2_134 with:= [[x], [c t1 t2]];

//assignLoc is an auxiliary relation for the first two hypotheses of rules 8 and 9
case x of [assignLoc c l e]:
  //update for rules 8 and 9
  W U:= {[assignLocType c l e t1] : [t1] in type1_2{[l]} | [assignLocType c l e t1] not in 0};
  assignLoc2_13 with:= [[l], [c e]];

//assignLocType is an auxiliary relation for assignLon and the third hypotheses of rules 8 and 9
case x of [assignLocType c l e t1]:
  //update for rule 8
  W U:= {[error c] : [t1] in type1_2{[c]} | not t2 <= t1, [error c] not in 0};
  assignLocType3_124 with:= [[e], [c l t1]];
  //update for rule 9
  W U:= {[error c] : [t2] in type1_2{[c]} | not t2 <= t1, [error c] not in 0};
  assignLocType1_234 with:= [[c], [l e t1]];

case x of [arralloc c a e1]:
  //update for rule 10
  W U:= {[arrallocArrenv c a e1 t1] : [t1 t2] in arrenv1_23{[a]} |
      [arrallocArrenv c a e1 t1] not in 0};
  arralloc2_13 with:= [[a], [c e1]];

//arrallocArrenv is an auxiliary relation for the first two hypotheses of rule 10
case x of [arrallocArrenv c a e1 t1]:
  //update for rule 10
  W U:= {[error c] : [t3] in type1_2{[e1]} | [error c] not in 0};
  arrallocArrenv3_124 with:= [[e1], [c a t1]];

case x of [arrassign c a e1 e2]:
  //update for rules 11 and 12
  W U:= {[arrassignArrenv c a e1 t1] : [t1 t2] in arrenv1_23{[a]} |
      [arrassignArrenv c a e1 t1] not in 0};
  arrassign2_134 with:= [[a], [c e1 e2]];

//arrassignArrenv is an auxiliary relation for the first two hypotheses of rules 11 and 12
case x of [arrassignArrenv c a e1 t1]:
  //updates for rule 11
  W U:= {[error c] : [t3] in type1_2{[e1]} | not t3 <= t1, [error c] not in 0};
  //update auxiliary map for rule 11
  arrassignArrenv2_134 with:= [[e1], [c a t1]];
  //updates for rule 12
  W U:= {[error c] : [t4] in type1_2{[e1]} | not t4 <= t1, [error c] not in 0};
  //update auxiliary map for rule 12
  arrassignArrenv1_234 with:= [[e2], [c a t1]];

case x of [sequence c c1 c2]:
  //update for rule 13
  W U:= {[type c1 t] : [t] in type1_2{[c]} | [type c1 t] not in 0};
  //update for rule 12
  W U:= {[type c2 t] : [t] in type1_2{[c]} | [type c2 t] not in 0};
  //update auxiliary map for rules 13 and 14
  sequence1_23 with:= [[c], [c1 c2]];

case x of [if c e c1 c2]:
  //update for rules 15 and 16
  W U:= {[ifType c c1 c2 t1] : [t1] in type1_2{[e]} | [ifType c c1 c2 t1] not in 0};
  //update auxiliary map for rules 15 and 16
  if2_134 with:= [[e], [c c1 c2]];

```

```

//ifType is an auxiliary relation for the first two hypotheses of rules 15 and 16
case x of [ifType c c1 c2 t1]:
  //update for rules 15 and 16
  W U:= {[type c1 t1] : [t2] in type1_2{[c]} | [type c1 Join(t1,t2)] not in 0};
  //update auxiliary map for rules 15 and 16
  ifType1_234 with:= [[c], [c1 c2 t1]];

case x of [while c e c1]:
  //update for rule 17
  W U:= {[whileType c c1 t1] : [t1] in type1_2{[e]} | [whileType c c1 t1] not in 0};
  //update auxiliary map for rule 17
  while2_13 with:= [[e], [c c1]];

//whileType is an auxiliary relation for the first two hypotheses of rule 17
case x of [whileType c c1 t1]:
  //update for rule 17
  W U:= {[type c1 t1] : [t2] in type1_2{[c]} | [type c1 Join(t1,t2)] not in 0};
  //auxiliary map for rule 17
  whileType1_23 with:= [[c], [c1 t1]];

case x of [letid c x e c1]:
  //update for rule 18
  W U:= {[type x t] : [t] in type1_2{[e]} | [type x t] not in 0};
  letid3_124 with:= [[e], [c x c1]];
  //update for rule 19
  W U:= {[type c1 t] : [t] in type1_2{[c]} | [type c1 t] not in 0};
  letid1_234 with:= [[c], [x e c1]];

case x of [type node t]:
  //update for auxiliary map used in rules 4 through 19
  type1_2 with:= [[node], [t]];
  //update for rule 4
  W U:= {[arithType e e2 t] : [e e2] in arith2_13{[node]} | [arithType e e2 t] not in 0};
  //update for rule 5
  W U:= {[type e t1] : [e t1] in arraccessArrenv2_13{[node]} | [type e Join(t1,t)] not in 0};
  //update for rule 6
  W U:= {[type e t1] : [e t1] in arithType2_13{[node]} | [type e Join(t1,t)] not in 0};
  //updates for rule 7
  W U:= {[assignIdType c x t] : [c x] in assignId3_12{[node]} | [assignIdType c x t] not in 0};
  W U:= {[assignIdTypeType node x t1 t] : [x t1] in assignIdType1_23{[node]} |
        [assignIdTypeType node node t1 t] not in 0};
  W U:= {[type node t1] : [c t1 t2] in assignIdTypeType2_134{[node]} |
        [type node Join(t1,t2,t)] not in 0};
  //update for rules 8 and 9
  W U:= {[assignLocType1 c node e t] : [c e] in assignLoc2_13{[node]} |
        [assignLocType1 c node e t] not in 0};
  //update for rule 8
  W U:= {[error c] : [c l t1] in assignLocType3_124{[node]} | not t <= t1, [error c] not in 0};
  //update for rule 9
  W U:= {[error c] : [l e t1] in assignLocType1_234{[node]} | [error c] not in 0};
  //updates for rules 11 and 12
  W U:= {[error c] : [c a t1] in arrallocArrenv3_124{[node]} | not t <= t2, [error c] not in 0};
  W U:= {[error c] : [c a t1] in arrassignArrenv2_134{[node]} | not t <= t1, [error c] not in 0};
  W U:= {[error c] : [c a t1] in arrassignArrenv1_234{[e1]} | not t <= t1, [error c] not in 0};
  //update for rule 13
  W U:= {[type c1 t] : [c1 c2] in sequence1_23{[node]} | [type c1 t] not in 0};
  //update for rule 14
  W U:= {[type c2 t] : [c1 c2] in sequence1_23{[node]} | [type c2 t] not in 0};

```

```

//updates for rules 15 and 16
W U:= {[ifType1 c c1 c2 t] : [c c1 c2] in if2_134{[node]} | [ifType1 c c1 c2 t] not in 0};
W U:= {[type c1 t1] : [c1 c2 t1] in ifType1_234{[node]} | [type c1 Join(t1,t)] not in 0};
W U:= {[type c2 t1] : [c1 c2 t1] in ifType1_234{[node]} | [type c2 Join(t1,t)] not in 0};
//updates for rule 17
W U:= {[whileType1 c c1 t] : [c c1] in while2_13{[node]} | [whileType1 c c1 t] not in 0};
W U:= {[type c1 t1] : [c1 t1] in whileType1_23{[node]} | [type c1 Join(t1,t)] not in 0};
//updates for rules 18 and 19
W U:= {[type x t] : [c x c1] in letid3_124{[node]} | [type x t] not in 0};
W U:= {[type c1 t] : [x e c1] in letid3_124{[node]} | [type c1 t] not in 0};

W less:= x;
if x not in input:
  0 with:= x;

```

Pseudocode for inferring maximum types of commands:

```

assign2_13 := {}; assignId2_13 := {}; assignLoc2_13 := {};
arralloc2_13 := {}; arrenv1_23 := {}; arrassign2_134 := {};
sequence2_13 := {}; sequenceHtype1_23 := {}; if3_124 := {}; ifHtype3_124 := {};
while3_12 := {}; letid4_123 := {}; htype1_2 := {}; type1_2 := {};
W := input U MIN;
0' := {};

while exists x in W:

  case x of [assign c var e]:
    //update for rule 20
    W U:= {[assignId c var e] | [assignId c var e] not in 0'};
    //update for rule 21
    W U:= {[assignLoc c var e] | [assignLoc c var e] not in 0'};
    //update to auxiliary map used in rules 20 and 21
    assign2_13 with:= [[var], [c e]];

  case x of [id x]:
    //update for rule 20
    W U:= {[assignId c x e] : [c e] in assign2_13{[x]} | [assignId c x e] not in 0'};

  //assignId is an auxiliary relation for the first two hypotheses of rule 20
  case x of [assignId c x e]:
    //update for rule 20
    W U:= {[htype c t] : [t] in type1_2{[x]} | [htype c t] not in 0'};
    //update to auxiliary map used in rule 20
    assignId2_13 with:= [[x], [c e]];

  case x of [loc l]:
    //update for rule 21
    W U:= {[assignLoc c l e] : [c e] in assign2_13{[l]} | [assignLoc c l e] not in 0'};

  //assignLoc is an auxiliary relation for the first two hypotheses of rule 21
  case x of [assignLoc c l e]:
    //update for rule 21
    W U:= {[htype c t] : [t] in r31typelt{[l]} | [htype c t] not in 0'};
    //update to auxiliary map used in rule 21
    assignLoc2_13 with:= [[l], [c e]];

  case x of [type node t]:
    //update for rule 20

```

```

W U:= {[htype c t] : [c e] in assignId2_13{[node]} | [htype c t] not in 0'};
//update for rule 21
W U:= {[htype c t] : [c e] in assignLoc2_13{[node]} | [htype c t] not in 0'};
//update to auxiliary map used in rules 20 and 21
type1_2 with:= [[node], [t]];

case x of [arralloc c a e1]:
//update for rule 22
W U:= {[htype c t2] : [t1 t2] in arrenv1_23{[a]} | [htype c t2] not in 0'};
//update to auxiliary map used in rule 22
arralloc2_13 with:= [[a], [c e1]];

case x of [arrenv a t1 t2]:
//update for rule 22
W U:= {[htype c t2] : [c e1] in arralloc2_13{[a]} | [htype c t2] not in 0'};
//update for rule 23
W U:= {[htype c t1] : [c e1 e2] in arrassign2_134{[a]} | [htype c t1] not in 0'};
//update to auxiliary map used in rules 22 and 23
arrenv1_23 with:= [[a], [t1 t2]];

case x of [arrassign c a e1 e2]:
//update for rule 23
W U:= {[htype c t1] : [t1 t2] in arrenv1_23{[a]} | [htype c t1] not in 0'};
//update to auxiliary map used in rule 23
arrassign2_134 with:= [[a], [c e1 e2]];

case x of [sequence c c1 c2]:
//update for rule 24
W U:= {[sequenceHtype c c2 t1] : [t1] in htype1_2{[c1]} | [sequenceHtype c c2 t1] not in 0'};
//update to auxiliary map used in rule 24
sequence2_13 with:= [[c1], [c c2]];

//sequenceHtype is an auxiliary relation for the first two hypotheses of rule 24
case x of [sequenceHtype c c1 t1]:
//update for rule 24
W U:= {[htype c t1] : [c2 t2] in htype1_2{[c1]} | [htype c Meet(t1,t2)] not in 0'};
//update to auxiliary map used in rule 24
sequenceHtype1_23 with:= [[c1], [c t1]];

case x of [if c e c1 c2]:
//update for rule 25
W U:= {[ifHtype c e c2 t1] : [t1] in htype1_2{[c1]} | [ifHtype c e c2 t1] not in 0'};
//update to auxiliary map used in rule 25
if3_124 with:= [[c1], [c e c2]];

//ifHtype is an auxiliary relation for the first two hypotheses of rule 25
case x of [ifHtype c e c2 t1]:
//update for rule 25
W U:= {[htype c t1] : [t2] in htype1_2{[c2]} | [htype c Meet(t1,t2)] not in 0'};
//update to auxiliary map used in rule 25
ifHtype3_124 with:= [[c2], [c e t1]];

case x of [while c e c1]:
//update for rule 26
W U:= {[htype c t] : [t] in htype1_2{[c1]} | [htype c t] not in 0'};
//update to auxiliary map used in rule 26
while3_12 with:= [[c1], [c e]];

case x of [letid c e x c1]:

```

```

//update for rule 27
W U:= {[htype c t] : [t] in htype1_2{[c1]} | [htype c t] not in 0'};
//update to auxiliary map used in rule 27
letid4_123 with:= [[c1], [c e x]];

case x of [htype node t]:
//updates for rule 24
W U:= {[sequenceHtype c c2 t] : [c c2] in sequence2_13{[node]}
      | [sequenceHtype c c2 t] not in 0'};
W U:= {[htype c t1] : [c t1] in sequenceHtype1_23{[node]} | [htype c Meet(t1,t)] not in 0'};
//updates for rule 25
W U:= {[ifHtype c e c2 t] : [c e c2] in if3_124{[node]} | [ifHtype c e c2 t] not in 0'};
W U:= {[htype c t1] : [c e t1] in ifHtype3_124{[node]} | [htype c Meet(t1,t)] not in 0'};
//update for rule 26
W U:= {[htype c t] : [c e] in while3_12{[node]} | [htype c t] not in 0'};
//update for rule 27
W U:= {[htype c t] : [c e x] in letid4_123{[node]} | [htype c t] not in 0'};
//update to auxiliary map used in rules 24 through 27
htype1_2 with:= [[node], [t]];

W less:= x;
if x not in input:
  0' with:= x;

```