On Access Control in Object-Oriented Database Systems

(オブジェクト指向データベースにおけるアクセス制御について)

田 島 敬 史

(1)

On Access Control in Object-Oriented Database Systems

by

Keishi Tajima
Department of Information Science
University of Tokyo

A Dissertation

Submitted to

The Graduate School of
The University of Tokyo
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Science
in Information Science

ABSTRACT

We develop fundamental mechanisms for access control in object-oriented database systems. In particular, we explore control depending on the static name of the accessed data (*name-dependent access control*), control depending on the dynamic values of the accessed data (*content-dependent access control*), and control depending on the context where that access occurs (*context-dependent access control*).

We achieve name-dependent access control by introducing multiple interfaces of objects, in other words multiple "views" of objects. In different views of an object, different sets of attributes are defined. We achieve content-dependent access control by introducing classes of objects defined with predicates. As a theoretical basis of those mechanisms, we develop a typed polymorphic calculus that supports general mechanisms for views on objects and for object sharing among classes [OT94]. In this calculus, a class contains inclusion specifications of objects from other classes. Each such specification consists of a *predicate* determining a subset of objects to be included and a *viewing function* through which those included objects are manipulated. Both predicates and viewing functions can be any type consistent programs definable in the polymorphic calculus. Inclusion specifications among classes can be cyclic, allowing mutually recursive class definitions. These features achieve flexible view definitions and wide range of class organizations in a compact and elegant way. The calculus provides a suitable set of operations for objects with views and classes with predicates so that the programmer can manipulate them just the same way as one deals with ordinary records and sets. Moreover, the calculus has a *type inference algorithm* that relieves the programmer from complicated type declarations of views and classes. The polymorphic type system of the calculus is also shown to be sound, which guarantees complete static check of type consistency of programs involving classes and views.

To achieve context-dependent control, we develop a framework of access control where the granularity of access control is an invocation of a composed function on data objects [Taj96]. The users are granted rights to invoke abstract functions instead of rights to invoke primitive operations. Although primitive operations are invoked inside those functions, the users can invoke them only through the granted functions, that is, only in some specific contexts. However, if the user effectively obtains the same ability that he would have when he could invoke those primitive operations directly, the functions actually do not hide the operations. Access control utilizing encapsulated functions easily causes many of this kind of "security flaws" through which malicious users can abuse the primitive operations inside the functions. We develop a formal basis to deal with this issue. First, we design a framework to describe *security requirements* that should be satisfied. In this framework, security requirements are described in terms of *inferability* on returned values of arbitrary functions and *controllability* on arguments on arbitrary functions. These two concepts are natural generalization of classical read and write capability, and are precise representation of user capability in database access. Next, in order to show the effectiveness of this framework, we develop an algorithm that syntactically analyzes program code of the functions and determines whether the given security requirements are satisfied or not.

論文要旨

オブジェクト指向データベースにおけるアクセス制御に関して，属性名などのデータの静的な名前に基づく制御 (名前依存制御)，データの値に基づく制御 (内容依存制御)，およびアクセスの文脈を考慮に入れる制御 (文脈依存制御) のための基本機構を開発する．

名前依存制御は各オブジェクトに複数のインタフェイス，すなわち複数のビューを定義し，ユーザ毎に異なるビューを提供することによって実現する．また，内容依存制御は，条件式によって定義されるオブジェクトのクラスを導入し，ユーザ毎に異なるクラスを提供することによって実現する．そのような枠組のための理論的な基礎として，オブジェクトのビューとクラス間のオブジェクト共有の機能を持つ多相型言語を設計する．この言語では，各クラス定義は他クラスとのオブジェクトの共有関係の定義を含む．この定義は共有すべきオブジェクトの集合を定める条件式と，それらのオブジェクトのインタフェイスを定義するビュー関数からなる．条件式とビュー関数には，この言語で定義可能な任意のプログラムを使用できる．また，相互再帰的クラス定義により，循環する共有関係も定義可能である．これらにより多彩なビュー定義やクラス構成が自然な形で記述できる．この多相型言語は型推論アルゴリズムを持ち，ユーザは繁雑なビューやクラスの型記述を行なう必要がない．このアルゴリズムは安全であり，これによりビューやクラスを含むプログラムの完全な静的型検査が可能である．

また文脈依存制御のために，データに対する高級関数の起動を制御の単位とするアクセス制御の枠組を開発する．各ユーザのアクセス権限は，基本操作ごとにではなく高級関数ごとに指定される．これらの高級関数の中では基本操作が起動されることになるが，ユーザはこれらの基本操作を与えられた高級関数を通してのみ，すなわち一定の文脈でのみ起動できる．しかし，もしこのユーザが，それらの基本操作を直接実行できるのと実質的には同じ能力を得るならば，これらの関数は基本操作を実質的には隠していないことになる．関数のカプセル化を利用してアクセス制御を行なう場合，このようなセキュリティの破れが容易に生じる．そこで我々はこのような問題を扱うための形式的な基礎を開発する．そのためにまず，セキュリティ上満たされるべき要求を記述するための枠組を設計する．この枠組では要求を「推論可能性」と「制御可能性」という概念を使って記述する．これらの概念は古典的な「読み出し」，「書き込み」の能力の一般化であり，ユーザのデータベースアクセスにおける能力を適切に表現している．そして次に，これらの概念が前述のようなセキュリティの問題を扱う上で有用であることを示すために，各関数のプログラムを静的に解析することによって，これらの概念を使って記述された要求が満たされているかどうかを判定するアルゴリズムを設計する．

Acknowledgements

Most of this research was done while the author was working under Atushi Ohori for two years at Research Institute for Mathematical Sciences (RIMS) in Kyoto University. I would like to thank Atsushi Ohori for his greatly valuable suggestions and long substantial discussions through the research. I also thank Kazuhiko Kato for his motivating me to beginning the research on views in object-oriented databases. The author is also indebted to Takashi Masuda for his continuous support and encouragement. Many other colleagues helped me in carrying out this research. Kenjiro Taura helped me in various things during the preparation of this thesis. Takashi Miyata continuously encouraged me so that I can concentrate on the work for this thesis. Their encouragement undoubtedly promoted the work. Masakazu Soshi helped me in surveying the research area on security issues. Naoki Kobayashi made a contribution to this research by giving a suggestion when the research on security analysis was in its early stage. The preparation of this thesis is carried out mainly at KABA. I would like to thank Reiji Nakajima for his allowing me to accessing and staying at KABA freely. Many colleagues at KABA, Yasuhiko Minamide, Koji Kagawa, Makoto Tanabe, Susumu Nisimura, Jun P. Furuse, Jacques Garrigue, Etsuko Suganami, and Minako Tarumi, provided me comfortable environment and pleasant relaxing time during the preparation of this thesis. I could have joyful time with them while I was working for this thesis.

Table of Contents

List of Figures

Chapter 1

Overview of The Thesis

We develop fundamental mechanisms for access control in object-oriented database systems. There are several classes of access control [FSW81]. Among them, *name-dependent access control*, which is control depending on the static name, such as attribute name, of the accessed data, and *content-dependent access control*, which is control depending on the dynamic value of the accessed data, are the most fundamental classes of access control. In the relational database systems, only these two classes of access control are usually supported. *context-dependent access control*, which is control depending on the context where that access occurs, is also widely regarded as a useful feature. In this thesis, we explore these three classes of access control.

This thesis consists of two parts. In Chapter 2, we discuss name-dependent access control and content-dependent access control. Name-dependent access control can be achieved by introducing multiple interfaces, in other words different "views", of objects, and by providing different views for different users. In different views, different set of attributes are defined. Each user can access only attributes defined in his view. On the other hand, content-dependent access control can be achieved by introducing classes of objects defined with predicates. Those classes dynamically include only objects that satisfy those predicates. By providing different sets of classes for different users, each user can access objects that satisfy the predicates of the classes given to him. By combining those two approaches, that is, by introducing classes with predicates and attaching different views to different classes, we can achieve both name-dependent access control and content-dependent access control. This approach is the traditional way to provide name-dependent and content-dependent access control in the relational database systems. "Views" in the relational databases are virtual relations defined by queries which specify conditions on the contents of tuples and attributes included in the view. Many recent researches also proposed introduction of views (or often called "virtual classes") similar to those in the relational databases into object-oriented database systems [TYI88, AB91, SLT91, Run92]. Those researches, however, introduce them into the ordinary object-oriented database systems in rather ad-hoc ways. Moreover, they just describe desired facilities, and no formal model of those facilities is established. The facilities needed for views are essentially those

for object-sharing among classes. In this thesis, we show that those facilities can be introduced as a natural generalization of the inheritance mechanism, which is one of the basic facilities of many object-oriented database systems. Further, as a theoretical basis of such a framework, we develop a typed polymorphic calculus that supports general mechanisms for object sharing among classes and views on objects.

In Chapter 3, we discuss context-dependent access control. To realize context-dependent control, we develop a framework of access control where the granularity of access to be controlled is an invocation of a composed function on data objects. Although primitive operations are invoked inside those functions, the users can invoke them only through the granted functions, that is, only in some specific contexts. The idea of access control using encapsulated functions is not new. In fact, this is exactly the idea of abstract data type, and is the important feature of many object-oriented database systems. Access control utilizing encapsulated functions, however, easily causes many "security flaws" through which malicious users can abuse the primitive operations inside the functions. In other words, it is difficult to ensure that a primitive operation in a function is invoked certainly in a desired context. In order to establish a formal basis to deal with this issue, we design a formal framework to specify a "desired context" in a more abstract way. Next, in order to show the effectiveness of our formalization, we develop techniques to statically detect security flaws, that is, to statically determine whether primitive operations are actually invoked in the specified "desired context". Because access control by providing for users only views defined by functions mapping original data to its view is essentially the same concept as access control by providing for users only composed functions, these techniques can also be applied to validation of the definitions of views in the framework shown in Chapter 2.

Chapter 2

A Polymorphic Calculus for Objects with Views and Classes with Object Sharing

2.1 Introduction

To provide name-dependent access control and content-dependent access control, we develop a mechanism to organize objects into classes in accordance with predicates on the dynamic values of those objects. In the usual object-oriented database systems, objects in the database are classified into classes. Each class has its extent, that is, a set of all objects that belongs to that class. Classes share objects in their extent with each other via *inheritance* mechanism. *Inheritance* is a term that implies hierarchically organizing some resources through a partial ordering on classes usually called an *IS-A* relation. As discussed in [ABD+89, BO91, BTBO91], this concept appears to be used for two different purposes in object-oriented databases. As in object-oriented programming languages in Smalltalk [GR83] tradition, it is used for *code sharing*; by asserting that Employee *IS-A* Person, we expect that any method applicable to Person objects is also applicable to Employee objects. In object-oriented databases, inheritance is also used for maintaining *extent inclusion* as mentioned above; by the assertion that Employee *IS-A* Person we usually assume that the set of Employee objects is a subset of the set of Person objects. Several object-oriented database systems and languages have been proposed by using these two forms of inheritance [BDK92, ACO85]. These two mechanisms called "inheritance" are, in essence, mechanisms for *sharing* of methods or objects among classes, but they are specialized for only cases which correspond to *IS-A* relation between classes. While this specialization provides a compact and intuitively appearing way of organizing methods and objects, there must be more fundamental framework for sharing. Indeed, there seems to be no a priori reason why each of these two kind of sharing should be controlled by simple partial orderings.

As for code sharing, *IS-A* relation achieves some flexibility of method application by allowing an object of one type to have its supertypes. This usage of *IS-A* relation can be regarded as an "approximation" to polymorphic typing in programming with records. Recent years have seen that *polymorphic type inference* for records provides a better alternative for method sharing [Wan87, JM88, OB88, Rém89]. In this paradigm,

the fact that Employee can inherit a method defined on Person corresponds to the property that the method is *polymorphic* so that it can also be applicable to elements of Employee. Since type inference directly captures the polymorphic nature of a method through inferring the principal type of the method, it achieves more general and rigorous model for method sharing. The *IS-A* relation is then regarded as a special case of method applicability represented by polymorphic typing for methods. As demonstrated by Machiavelli [OBBT89, BO96], this paradigm can be successfully applied to database programming for various forms of complex objects.

We believe that a similar situation exists for object sharing. Extent inclusion through *IS-A* relation is certainly useful for many cases. However, there are others where a simple partial ordering is inadequate for expressing the desired structure of object sharing. For example, suppose we have classes Student and Employee and want to define a new class FemaleMember. We naturally want the class FemaleMember to include Student objects and Employee objects that are considered also as female. Such a situation cannot be directly modeled by usual inclusion through a partial ordering. There are some proposals [TYI88, SLT91, AB91, Run92] that attempt to overcome this limitation by introducing special classes that are dynamically derived from other classes in accordance with the specifications of them, and by placing them in a usual inheritance hierarchy. This process, however, generally requires the generation of intermediate classes to conform to the presupposed model of extent inclusion. In addition, the treatment of update to such dynamically derived classes needs ad-hoc rules. However, we need not be bound by partial ordering. A more direct and natural approach is to develop a framework for object sharing where the programmer can specify desired object sharing relations between arbitrary two classes and with an arbitrary condition. Usual extent inclusion, where one class unconditionally includes the entire extent of another class, then becomes a special case of this general mechanism. The motivation of our work is to develop such a framework.

The first step in our development is to define a flexible mechanism for *views* of individual objects. The ability for a class to share objects with another class necessarily implies the ability to view an object differently depending on the context in which the object is manipulated. For example, if FemaleMember class imports some Employee object, then the imported Employee object should behave like a FemaleMember object when evaluating a query on FemaleMember class. We realize views of objects by attaching an unevaluated function, which we call *viewing function*, onto every object. The viewing function would be dynamically evaluated and maps the raw attribute information of that object to the view. Any type consistent program can be used as a viewing function. Users can manipulate an object as if it has the type of the range of the viewing function, which we call the *view type* of that object. In addition, we can attach some view accumulatively on top of another existing view. Internally, that is realized by function composition.

The idea of representing views through functions is not new. In the relational model, a view is essentially an expression specifying how the relations be transformed. Gottlob, Paolini and Zicari [GPZ88] developed a

theory of relational view update based on the notion of *dynamic views*, which is regarded as an association of a data object and a viewing function. In the context of object-oriented databases, Heiler and Zdonik [HZ90] uses a similar notion to provide a viewing mechanism. "View object" in [Wie86, BKSW91] can also be regarded as an unevaluated viewing function. The general idea of views is also related to "roles" of objects [RS91, ABGO93]. These proposals contain a mechanism to attach multiple roles to objects, which can be regarded as a simple method to implement some aspect of views. However, these existing approaches in object-oriented databases only describe some desired features of views operationally; there seems to be no formal framework for systematic manipulation of objects and views with a rigorous semantics. Here we attempt to provide such a framework within a paradigm of type theory of programming languages. We hope that the framework presented here will serve as a typed foundation for statically typed polymorphic object-oriented database programming languages.

The second step of our development is to define a mechanism for object sharing among classes. We allow a class definition to contain inclusion specifications of objects from other classes. Therefore, the extent of each class consists of its own extent and imported extent which is dynamically calculated. Each such inclusion specification consists of a *predicate* that selects the subset of objects to be included and a *viewing function* defining view under which the included objects are manipulated. Both predicates and viewing functions can be any type consistent programs definable in the polymorphic calculus. This achieves flexible sharing among classes. Moreover, the dependence structure among classes induced by the inclusion specifications can be cyclic, and therefore *mutually recursive class definitions* are possible. Cyclic sharing relation are necessary typically for mutual sharing between two classes. For example, in the example above, if someone inserts an instance to class Student and that instance is a female, then class FemaleMember should include that instance while if someone inserts an instance to class FemaleMember and that instance is a student, then that instance should be also included to class Student. Allowing any recursive definitions, however, can cause ambiguous definitions or infinite computations. In this development, we give proper primitives for class definition so that no such problems occurs and we define clear semantics for those primitives.

One more point to note in our development is that we integrate those two mechanisms for object sharing into the polymorphic type inference system for method sharing. Thus, we provide a general and formal basis for object-oriented databases including both method sharing and object sharing. Our development of such a polymorphic language for views and classes is carried out in a following three steps. We first define a polymorphic language for labeled records and sets similar to Machiavelli [OBBT89, BO96] as the core language. We then define the necessary structures and operations for views on top of the core language. In order to define a framework for uniform manipulation of objects with same view type but with different internal structure, we should define a proper set of operations that allow the programmer to manipulate objects just the same way as one manipulates ordinary values of the view type of those objects. This uniformity is essential for extending the language for objects to classes, which correspond to collections

of those objects that seem to have the same view type but may have different data structure of attribute information. In this development, we give such a set of operations and define their precise semantics and an effective implementation algorithm by developing a systematic method to translate the operations into the core calculus. Finally, we develop a mechanism for classes with general sharing relations on top of the language for objects. A class is conceptually represented as a pair consisting of a set of objects representing its *own extent* and an *inclusion function* that computes a set of objects included from other classes. Similar to what we do for objects, we define a set of operations on those classes in such a way that the programmer can manipulate classes just the same way as one manipulate usual sets of objects. We then define their precise semantics and an effective implementation algorithm by developing a systematic method to translate the operations into the language we have defined for objects. We show that the two translations we described preserve typings, which establishes that the language with views and classes can be statically type-checked. Moreover, there is an algorithm to infer a principal type for any type consistent program involving views and classes.

Views and object sharing have also been considered in deductive approaches, where views are represented by a form of deductive rules. Based on this general idea, in [KLW90, ALUW93], a semantically sound accounts for views are given. These approaches seems to depends crucially on the properties of the underlying framework, i.e. formal logic. It is not at all clear how the notions of views and objects in deductive framework are related to those in database programming languages. Precise comparison of our approach with those in deductive databases is certainly interesting, but it may require a more abstract characterization of views and objects. We will comment more on this in Section 2.6.

One issue we have not addressed in this thesis is a proper treatment of persistent data, which require some form of dynamic typing. Connor *et. al.* [CDMB90] demonstrated that various features of views of persistent data can be represented by combining a form of localized dynamic typing and existential types. Their techniques seems to be complementary to our method.

The rest of this chapter is organized as follows. Section 2.2 defines the core language for records and sets. Section 2.3 extends the core to objects. Section 2.4 extends the language to classes. Section 2.5 describe some examples of applications of our calculus. Section 2.6 discusses some further issues and concludes this part.

2.2 The Core Language

This section defines a polymorphic calculus similar to Machiavelli. This serves as the core language on which view and object sharing mechanisms will be developed later. We first explain two important data structures: record and set.

The syntax for records is:

$$[f_1 = \ldots, f]$$

where f denotes *fields* whose syntax is either $l = e$ for immutable fields or $l := e$ for mutable fields. For example,

```
let joe = [Name = "Doe", Salary := 3000]
```

will yield a record with immutable Name field and mutable Salary field. In most cases, it is this structure that store attribute information in objects. To properly capture mutability of objects, we decide that evaluation of a record expression creates a new identity (internally implemented by a reference) and that equality on records is identity, i.e. L-value equality.

There are three operations on records. The first is field extraction: $r{\cdot}l$, which extracts the value of l field from the record r. The extracted value is always an ordinary value, i.e. R-value even if the l field in r is mutable. The second is a special form of field extraction: extract(r, l), which extracts the L-value of the mutable field l of the record r. The extracted L-values can only be used as field values in record creations. For example,

```
let doe = [Name = "Doe", Income := extract(joe, Salary)]
let john = [Name = "John", Salary = extract(joe, Salary)]
```

are legal, resulting in joe's Salary field, doe's Income field, and john's Salary field all sharing the same L-value. If one changes joe's Salary field, then that change will be reflected to doe and also to john even though john's Salary field is immutable. However, both of the following are illegal and will be rejected by the type system we shall define later:

```
[Name = "Joe Doe", Income = extract(joe, Salary) * 2]
[Name = extract(joe, Name), Income := joe·Salary]
```

The first try to perform arithmetic operation on an extracted L-value and the second attempts to extracts the L-value of an immutable field. Distinguishing these two forms of field extraction allow us to properly transfer mutability of fields to views. The third operation on records is field update: update(r, l, e), which changes the value of the mutable l field of the record r to e. For example, update(joe, Salary, 4000) will change joe's Salary value to 4000, while update(joe, Name, "Peter") is illegal and is rejected by the type system, since joe's Name field is immutable.

Another data structure we consider here is set, whose syntax is:

$$\{e_1, \ldots, e_n\}$$

The basic operations for sets are: union(e, e) and hom(S, f, op, z). hom [OBBT89] is a general iteration operation similar to "pump" in FAD[BBKV88]. Its intuitive meaning can be explained by the equation:

$$\mathsf{hom}(\{e_1, \ldots, e_n\}, f, op, z) = op(f(e_1), op(f(e_2), \ldots, op(f(e_n), z)\ldots))$$

There are other possibilities for a general elimination operation for sets. We believe that adoption of a different elimination operation for sets will not affect the mechanisms for views and classes we shall develop in this thesis. The following operations are definable using union and hom:

$$\begin{aligned}
\mathsf{member}(e_1, e_2) &: \text{test whether } e_1 \text{ is a member of } e_2 \\
\mathsf{remove}(e_1, e_2) &: \text{remove an element } e_1 \text{ from a set } e_2 \\
\mathsf{prod}(e_1, \ldots, e_n) &: \text{return the } n\text{-ary Cartesian product of sets } e_1, \ldots, e_n \\
\mathsf{map}(e_1, e_2) &: \text{map a function } e_1 \text{ to a set } e_2 \\
\mathsf{filter}(e_1, e_2) &: \text{select all the elements } x \text{ in a set } e_2 \text{ such that } (e_1\ x) = \mathsf{true}
\end{aligned}$$

These operations will be used later for the development of views and classes.

By integrating records and sets in a lambda calculus, we define the syntax of the core language:

$$\begin{aligned}
e \quad ::= \quad & c^\tau \mid () \mid x \mid \mathsf{eq}(e, e) \mid \lambda x.e \mid (e\ \ e) \mid [f, \ldots, f] \mid e{\cdot}l \mid \mathsf{extract}(e, l) \mid \mathsf{update}(e, l, e) \mid \\
& \{e, \ldots, e\} \mid \mathsf{union}(e, e) \mid \mathsf{hom}(e, e, e, e) \mid \mathsf{fix}\ x.e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e\ \mathsf{end}
\end{aligned}$$

c^τ stands for constants of type τ. () is the only value of type *unit* that can be returned by functions such as update. x stands for variables of the calculus. $\mathsf{eq}(e_1, e_2)$ is equality test. For records and functions, eq uses L-value equality; for other types, it uses the usual value equality. $\lambda x.e$ stands for lambda abstraction, and $(e\ \ e)$ for function application. fix $x.e$ is for recursive function definition where x can occur free in e, and let $x = e$ in e end is ML's polymorphic let construction. By combining fix, let, lambda abstraction, and record, it is possible to define the following mutually recursive function definition:

$$\mathsf{fun}\ f_1\ x_1 = e_1\ \mathsf{and}\ \cdots\ \mathsf{and}\ f_n\ x_n = e_n$$

where the functions $f_1, \ldots, f_n$ being defined may be used in the bodies $e_1, \ldots, e_n$ of those function definitions. We use pairs $\langle e_1, e_2 \rangle$ as an abbreviation for two element records with numeric labels, and the projections $e{\cdot}1$ and $e{\cdot}2$ for the corresponding field extractions. Accordingly, we write $\tau_1 \times \tau_2$ for $[1 = \tau_1, 2 = \tau_2]$. We also write $\lambda().e$ for a function whose domain type is *unit*.

The type system of the language is an adaptation of that of [Oho92] with a refinement for distinguishing immutable and mutable record fields. The set of monotypes (ranged over by τ) of the language is given by the syntax:

$$\tau ::= b \mid \mathit{unit} \mid t \mid \tau{\rightarrow}\tau \mid \{\tau\} \mid \mathcal{L}(\tau) \mid [F, \ldots, F]$$

where b stands for base types such as *string*, t for type variables, $\tau{\rightarrow}\tau$ for function types, $\{\tau\}$ for set types whose element type is τ. $\mathcal{L}(\tau)$ is a type of L-values of a field of type τ in a record. L-values can be used as first class values. They can be passed from functions to functions as their arguments or their return values. However, we can not manipulate the contents of L-values. The only substantial usage of L-values is to use them as field values in record creations. $[F, \ldots, F]$ stands for record types where F is either $l = \tau$ for immutable fields or $l := \tau$ for mutable fields.

To represent polymorphic types for operations on records, we place *kind constraint* on type variables. The set of kinds (ranged over by K) is given by the grammar:

$$K ::= U \mid [\![F, \ldots, F]\!]$$

U denotes arbitrary types, while $[\![F_1, \ldots, F_n]\!]$ denotes those record types that contain fields $F'_1, \ldots, F'_n$ and possibly others, where each F'_i must satisfy the following condition: if F_i is $l := \tau$ then F'_i must be $l := \tau$, on the other hand if F_i is $l = \tau$ then F'_i can be either $l = \tau$ or $l := \tau$. We write $F_i < F'_i$ for this condition. We write $t :: K$ for a type variable t with a kind constraint K. For such a type variable, only those types having the kind K can be substituted. Using kinds, the set of polytypes is defined by the syntax:

$$\sigma ::= \tau \mid \forall t :: K.\sigma$$

$\forall t :: K.\sigma$ is a polymorphic type where t is quantified over the subset of types denoted by the kind K.

The type system of this language has two forms of judgements. A kinding judgement $\mathcal{K} \vdash \tau :: K$ asserts that the type τ has the kind K under the kind assignment $\mathcal{K}$, which is a mapping from type variables to kinds. A typing judgements $\mathcal{K}, \mathcal{A} \rhd e : \sigma$ asserts that the expression e has the type σ under $\mathcal{K}$, and a type assignment $\mathcal{A}$, which is a mapping from variables to types. Figure 2.1 and Figure 2.2 shows the rules to derive these two forms of judgements. Note that the mutability and immutability of the field is correctly enforced by restricting field l in the rules for special field extraction and field update to be mutable field, while that in the rule for field selection can be immutable field. Also note that the rule for record is the only rule that eliminates $\mathcal{L}(\tau)$.

Since ML-style polymorphic typing is not sound with respect to the usual operational semantics of mutable values [Mil78], we place the restriction that the type of mutable fields be ground monotypes. With this restriction, we can show that the type system is sound with respect to an operational semantics of the language in the style of [Oho92], and that "well typed programs cannot go wrong" as shown by Milner [Mil78] for ML. Moreover, there is an algorithm to compute a principal type of any type consistent program. The following properties can be shown by the techniques developed in [Oho92].

Proposition 1 *If $\mathcal{K}, \mathcal{A} \rhd e : \tau$ and e evaluates to v under an environment that respects the kind assignment and the type assignment $\mathcal{K}, \mathcal{A}$, then v has the type τ.*

Proposition 2 *For any e and $\mathcal{K}, \mathcal{A}$, if e has a typing under $\mathcal{K}$ and $\mathcal{A}$, then e has a principal type under $\mathcal{K}, \mathcal{A}$ such that any other type under $\mathcal{K}, \mathcal{A}$ is its instance. Moreover, a principal type is computed effectively.*

For example, we can infer the principal type of the following function including record structure and set structure as below:

$$\begin{aligned}&\mathsf{let\ wealthy} = \lambda \mathsf{S.map}(\lambda \mathsf{x.x{\cdot}Name}, \mathsf{filter}(\lambda \mathsf{x.(x{\cdot}Salary} > 100), \mathsf{S}))\\ &\quad :\ \forall t_1 :: U.\ \forall t_2 :: [\![\mathsf{Name} = t_1, \mathsf{Salary} = \mathit{int}]\!].\ \{t_2\} \rightarrow \{t_1\}\end{aligned}$$

$\mathcal{K} \vdash \tau :: U$ for all τ

$\mathcal{K} \vdash t :: [\![F_1, \ldots, F_n]\!]$ if $t \in domain(\mathcal{K}), \mathcal{K}(t) = [\![F'_1, \ldots, F'_n, \ldots]\!]$ such that $F_i < F'_i (1 \leq i \leq n)$

$\mathcal{K} \vdash [F'_1, \ldots, F'_n, \ldots] :: [\![F_1, \ldots, F_n]\!]$ such that $F_i < F'_i (1 \leq i \leq n)$

(const) $\mathcal{K}, \mathcal{A} \rhd c^\tau : \tau$

(unit) $\mathcal{K}, \mathcal{A} \rhd () : unit$

(var) $\mathcal{K}, \mathcal{A} \rhd x : \sigma$ if $x \in domain(\mathcal{A}), \mathcal{A}(x) = \sigma$

(eq) $$\frac{\mathcal{K}, \mathcal{A} \rhd e_1 : \tau \qquad \mathcal{K}, \mathcal{A} \rhd e_2 : \tau}{\mathcal{K}, \mathcal{A} \rhd \mathsf{eq}(e_1, e_2) : bool}$$

(abs) $$\frac{\mathcal{K}, \mathcal{A}\{x \mapsto \tau_1\} \rhd e_1 : \tau_2}{\mathcal{K}, \mathcal{A} \rhd \lambda x.e_1 : \tau_1 \rightarrow \tau_2}$$

(app) $$\frac{\mathcal{K}, \mathcal{A} \rhd e_1 : \tau_1 \rightarrow \tau_2 \qquad \mathcal{K}, \mathcal{A} \rhd e_2 : \tau_1}{\mathcal{K}, \mathcal{A} \rhd (e_1 \ e_2) : \tau_2}$$

(record) $$\frac{\mathcal{K}, \mathcal{A} \rhd e_1 : \tau'_1, \ \ldots \ , \mathcal{K}, \mathcal{A} \rhd e_n : \tau'_n}{\mathcal{K}, \mathcal{A} \rhd [l_1 @_1 e_1, \ldots, l_n @_n e_n] : [l_1 @_1 \tau_1, \ldots, l_n @_n \tau_n]}$$ where $@_i$ is $=$ or $:=$, and τ'_i is τ_i or $\mathcal{L}(\tau_i)$

(dot) $$\frac{\mathcal{K}, \mathcal{A} \rhd e : \tau_1 \qquad \mathcal{K} \vdash \tau_1 :: [\![l = \tau_2]\!]}{\mathcal{K}, \mathcal{A} \rhd e.l : \tau_2}$$

(extract) $$\frac{\mathcal{K}, \mathcal{A} \rhd e : \tau_1 \qquad \mathcal{K} \vdash \tau_1 :: [\![l := \tau_2]\!]}{\mathcal{K}, \mathcal{A} \rhd \mathsf{extract}(e, l) : \mathcal{L}(\tau_2)}$$

(update) $$\frac{\mathcal{K}, \mathcal{A} \rhd e_1 : \tau_1 \qquad \mathcal{K}, \mathcal{A} \rhd e_2 : \tau_2 \qquad \mathcal{K} \vdash \tau_1 :: [\![l := \tau_2]\!]}{\mathcal{K}, \mathcal{A} \rhd \mathsf{update}(e_1, l, e_2) : unit}$$

Figure 2.1: Type System (1/2)

(set) $$\frac{\mathcal{K},\mathcal{A} \rhd e_i : \tau \quad (1 \leq i \leq n)}{\mathcal{K},\mathcal{A} \rhd \{e_1,\ldots,e_n\} : \{\tau\}}$$

(union) $$\frac{\mathcal{K},\mathcal{A} \rhd e_1 : \{\tau\} \qquad \mathcal{K},\mathcal{A} \rhd e_2 : \{\tau\}}{\mathcal{K},\mathcal{A} \rhd \mathsf{union}(e_1,e_2) : \{\tau\}}$$

(hom) $$\frac{\mathcal{K},\mathcal{A} \rhd e_1 : \{\tau_1\} \quad \mathcal{K},\mathcal{A} \rhd e_2 : \tau_1 \to \tau_2 \quad \mathcal{K},\mathcal{A} \rhd e_3 : \tau_2 \times \tau_3 \to \tau_3 \quad \mathcal{K},\mathcal{A} \rhd e_4 : \tau_3}{\mathcal{K},\mathcal{A} \rhd \mathsf{hom}(e_1,e_2,e_3,e_4) : \tau_3}$$

(fix) $$\frac{\mathcal{K},\mathcal{A}\{x:\tau\} \rhd e : \tau}{\mathcal{K},\mathcal{A} \rhd \mathsf{fix}\ x.e : \tau}$$

(let) $$\frac{\mathcal{K},\mathcal{A} \rhd e_1 : \sigma \quad \mathcal{K},\mathcal{A}\{x \mapsto \sigma\} \rhd e_2 : \tau}{\mathcal{K},\mathcal{A} \rhd \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : \tau}$$

(gen) $$\frac{\mathcal{K}\{t \mapsto k\},\mathcal{A} \rhd e : \sigma}{\mathcal{K},\mathcal{A} \rhd e : \forall t :: k.\sigma} \quad t \text{ not free in } \mathcal{A}$$

(inst) $$\frac{\mathcal{K},\mathcal{A} \rhd e : \forall t :: k.\sigma \quad \mathcal{K} \vdash \tau :: k}{\mathcal{K},\mathcal{A} \rhd e : \sigma[\tau/t]}$$

Figure 2.2: Type System (2/2)

This type indicates that this function can be applied to any set of records with at least Name field of arbitrary type and Salary field of type *int*, and returns a set of the type of Name field. Suppose there are two sets defined as below:

let managerSet = $\{\ldots\}$: $\{[\mathsf{Name} = string, \mathsf{Age} := int, \mathsf{Salary} := int, \mathsf{Section} := string]\}$
let employeeSet = $\{\ldots\}$: $\{[\mathsf{Name} = string, \mathsf{Age} := int, \mathsf{Salary} := int, \mathsf{Job} := string]\}$

Then, both of the following function application is determined as valid by the type system:

(wealthy managerSet) : $\{string\}$
(wealthy employeeSet) : $\{string\}$

2.3 View Extension

This section extends the core with a mechanism for manipulating objects with views. As explained in the introduction, we realize a view of an object by attaching a viewing function onto each object, which translates the raw attribute information stored in that object to the view. That viewing function is applied to the attribute information immediately before some query is performed on that object, and then the query is actually performed on the result of that application. By that, users can manipulate the object as if manipulating the usual value of the type of that view.

Conceptually we regard an object as a pair of a viewing function and a raw attribute information. There are several ways to implement it. One way is to implement an object directly as a pair of a function that work as a viewing function and a record, which we call *raw object* and works as a carrier of an identity and attribute information. A new object is created by specifying a record used as a raw object. Equality on objects is determined based on the equality on raw objects, and therefore objects created with the same raw object have the same identity. When a query is issued on that object, the viewing function is applied to the raw object and the raw object is mapped to the view type, and then the query is executed on the result of the mapping. Another way is to implement an object as a *state function* associated with an *object identifier*. The state function is a function closure which encapsulates both the attribute information and the viewing function of that object. The contents of the state function is simply applying the viewing function to the attribute information and returning the result of that application. The evaluation of the state function is delayed. In the same way as in the another way of implementation, it would be evaluated each time some query is issued on that object. The object identifier is used to determine the equality on objects independent from the difference of their views. Each time a new object is created, a new identifier is assigned to it, and from that time on, the identifier shall not be changed even after a new view is defined on that object. Each of those two approaches has its advantages and disadvantages. One advantage of the former way is that it can be implemented even with the language that does not support function closure, such as C language. With C language, an object would be implemented as a pair of a pointer to a function

and a record structure. On the other hand, one advantage of the latter way is that we can define a view that augment the information of an object as explained later. In this thesis, we choose the latter approach. Even if we chose the former approach, however, the similar development and the proof of the soundness of the polymorphic type system would be possible.

Our object supports encapsulation. Detail of the data structure of attribute information is hidden to the users and they can access objects only through the views. Outside objects, the users can define functions that access objects through the views, and we have a mechanism of code sharing for these user functions. In some sense, this structure can be regarded as corresponding to the well-known three-level architecture of database schema [ANS75]; internal level that is the physical implementation of data, conceptual level that is the well-defined interface of data to users, and external level where users can freely define customized interface appropriate for their application. Attribute information corresponds to the physical structure level, view corresponds to the conceptual interface level, and user functions correspond to the external level.

This structure seems to be different from usual object models for object-oriented databases. In many object-oriented database systems, an object belongs to some classes and the object can be manipulated only by invoking methods defined on those classes. Those methods can be shared among classes by inheritance. If any of usual users is allowed to define those methods, it means objects are not encapsulated to the user. On the contrary, if usual users are only allowed to define functions outside of the classes, objects are encapsulated but those users cannot utilize the inheritance in their defining user functions. In other object-oriented databases that use polymorphic type inference for code sharing, an object is just a bare record, and the mechanisms for code sharing works for functions directly accessing those records. Again, in this case, if any user can define those functions, objects are not encapsulated, and if usual users cannot define those functions, they cannot utilize the mechanisms for code sharing.

On the other hand, in our model, code sharing works for user's functions that can access objects only through their interface. To provide code sharing for those user's functions, we develop a type system that can deal with polymorphic nature of functions manipulating objects through their views, by properly lifting a polymorphic type system for usual record functions. In addition, while in the usual models interface of objects is defined at classes, in our model, individual objects have their own interface. By that, as explained in a later section, classes in our model can be sets of objects with same conceptual interface, i.e. with same view type, but with heterogeneous internal structure, while classes in usual models are sets of objects with homogeneous internal structure. This heterogeneity is especially important for databases, because database objects are persistent and it cause several issues called schema versioning, heterogeneous database, and so on. Moreover, as we mentioned in the introduction, this heterogeneity is essential for object sharing among classes we develop in this research. Although we do not discuss more on these issues except for object sharing, we believe objects for databases should have this structure for these various issues.

2.3.1 An algebra for objects and views

The set of expressions is extended with the following set of expression constructors, which serves as an "algebra" for objects and views:

$$e ::= \cdots \mid \mathsf{obj}(e) \mid (e \text{ as } e) \mid \mathsf{query}(e,e) \mid \mathsf{fuse}(e,e) \mid \mathsf{relobj}(l_1 = e_1, \ldots, l_n = e_n)$$

The intended meanings of these constructors are explained below. Their precise semantics is given later by defining a translation of them into the core language.

Object creation: $\mathsf{obj}(e)$.

This construct takes some value, in most cases a record e. Then, it creates a new object by using e as its raw attribute information. The created object is an association of the state function that simply returns that raw attribute information and a fresh identifier.

View composition: $(e_1 \text{ as } e_2)$.

Given an object e_1 and a function e_2, this creates a new object whose identifier is the same as that of e_1 and whose state function is the composition of that of e_1 and the function e_2. That is, it creates a new object by attaching a view defined by e_2 on top of the object e_1. Although this construct creates a new object with a new view, the created object has the same identity as the original object e_1 because identity of objects is determined by their identifier as explained before. After this view definition, one can manipulate the created object as if it has the type of the range of e_2, which we call *view type* of that object. The detail of manipulation of objects with views will be explained later.

By this construction, we obtain the desired closure property of objects and view definition; defining a view on an object yields another object that has the same formal status as being an object, and therefore yet another views can freely be defined on it. In addition, in that definition, we can use functions whose domain is the view type of that object. It means that even when defining views on objects, we can see those objects through their views and need not know the internal structure of the attribute information.

We do not usually regard a function that changes the state of an object as a viewing function. So it would be useful for the type system to check whether e_2 in this construct updates any mutable field of some record included in attribute information. It seems to be not hard to refine the type system to check whether e_2 involves `update` operations (directly or indirectly through other functions called from e_2). However, this significantly increases the complexity of the type system, and is not dealt with here. Also, there might be a case where a function with side effect can be considered as a suitable viewing function.

Query on views: $\mathsf{query}(e_1,e_2)$.

This evaluates a query specified by a function e_1 against an object e_2. It is only this operation that actually evaluates a viewing functions of objects. Intuitively, this expression first evaluates or "materializes" the view by evaluating the state function of e_2, and then applies the function e_1 to the result of that evaluation. e_1 can be any function applicable to e_2's view type. Therefore, using this construct, we can freely manipulate the object as if it has its view type. In most cases, the view type of objects may be some

record type and we can use usual record functions. We need not worry about writing complicated interface functions for properly lifting ordinary functions on records to views. As a special case of this construction, if e_1 is the identity function, then this construct simply returns the current value under the view.

Generalized equality on objects: fuse(e_1,e_2).

Identity of objects is determined by their identifier. If two object have different state functions and therefore different view type but they have the same identifier, they have the same identity. On the contrary, even if two object share the same attribute information, if they have different identifiers, it means they are created in different object creations, therefore those objects have different identity. Notice that by the term "identity", we are not referring to the special characteristic that distinguishes mutable objects from usual values. Here, the term "identity" simply means the equality in the sense that "those objects are regarded by the users as representing the identical entity in the real world". In database, mechanisms to express such user's intention is important, because we often need to represent one real-world entity by multiple database objects in order to describe various views, aspects, or versions of that entity.

This construct first tests whether e_1 and e_2 have the same identifier, i.e. the same identity. If they do then it returns a singleton set of a new object whose identifier is the same as that of e_1 and e_2, and whose view encodes the views of both e_1 and e_2 in a pair. Therefore, its view type is a product of view types of e_1 and e_2. If e_1 and e_2 have different identifier, then the result is the empty set. This operation is inspired by a similar operation considered in [BO96], and is regarded as a generalization of equality test for objects. It can also be considered as a translation of equality-join for relations into the context of our objects.

Relation object creation: relobj($l_1 = e_1, \ldots, l_n = e_n$).

This creates a relation object from the given set of objects. Different from all the previous operations, this creates a new object identifier, and therefore new identity. The new view is composed of the views of $e_i, \ldots, e_n$, and corresponds to the relation of their views. Combining with sets, this construction can be used to represent relation objects between sets of objects.

Although we have not yet developed a formal basis to discuss expressive power of languages for objects and views, we believe that the operations just defined form a sufficient set of operations for manipulating objects and views; any other operations for objects and views seem to be definable using these together with lambda abstraction. Some useful operations for objects and views definable using these five operations include:

- Equality test for objects: objeq(e_1, e_2). This tests whether two objects has the same identifier and therefore have the same identity. This is implemented as

  not(eq(fuse(e_1,e_2), {}))

- Select view mapping: (select as e from S where p). This selects those objects that satisfies p from S with attaching a new view defined by a function e. This is implemented as:

  map(λx.(x as e), filter(p,S))

- Intersection: intersect(e_1, e_2). Given sets e_1 and e_2 of objects, this returns a new sets that corresponds to the intersection of the two. The view type of objects of the resulting set is the product type of the view types of objects of the two sets. This is implemented as:

  hom(prod(e_1,e_2), λx.fuse(x·1,x·2), union, {})

  It is easy to generalize this to n-ary intersection intersect($e_1, \ldots, e_n$), which will be used in the following development.

- Relation style query: relation $[l_1{=}e_1, \ldots, l_n{=}e_n]$ from $x_1 \in S_1, \ldots, x_m \in S_m$ where p where e_i and p can contain x_i. This creates a set of relation objects satisfying the predicate p from the sets S_i of objects. This can be implemented by lifting the techniques we used in Machiavelli [BO96] to objects and views using relation object creation. One implementation is the following:

  map(λx.x·1,
  filter(λy.y·2,
  map($\lambda(x_1, \ldots, x_m).\langle$relobj($l_1 = e_1, \ldots, l_n = e_n$), $p\rangle$,
  prod($S_1, S_2, \ldots, S_m$))))

The last three involve sets of objects. Before we proceed, there is one decision to be made on the semantics of sets of objects. Since there are two form of equality on objects, we need to determine which equality is used in formation of a set of objects. Here we choose objeq. This decision yields a simpler formulation of recursive classes and views we shall develop later.

Choosing objeq implies that we need to "collapse" elements that are objeq whenever we make a union of two sets. Conceptually, there would be two alternatives. One is to require that if a union of two sets contains two objects that are objeq, then they must also have the same state function, i.e. the function closures with the same L-value. The other alternative is to select one object among those that have the same identity. Here we choose the latter approach and assume that if $e_1 \in S_1$ and $e_2 \in S_2$ such that objeq(e_1, e_2), then $S_1 \cup S_2$ will choose e_1 and discard e_2. This will yield a more flexible mechanism for classes and object sharing. However, the other alternative is equally possible. The following development can easily be adopted for the other semantics of sets of objects.

One considerable alternative in selecting primitives is to provide primitives by which users can directly manipulate identifiers of objects. For example, instead of providing one primitive obj, we can provide the following primitives: (1) id() which creates a fresh identifier whenever evaluated, (2) idextract(e) which extracts the identifier of the object e, and (3) obj(e_1,e_2) which creates a new object using e_1 in the same way as obj, but which assigns the identifier e_2 to that object. In the situations where we need to describe multiple versions of the same entity, the ability to explicitly assign an identifier of some existing object to some newly created object is needed. In this development, however, we do not choose this alternative

because here we treat only view mechanisms, and in that context we think we should not be able to directly manipulate identifiers.

One important feature of our approach is that such viewing mechanisms are typed operations in our polymorphic type system. We introduce a new type constructor $obj(\tau)$ for objects with view type τ, in other words, objects that can be manipulated as if it has type τ. As seen below, this type is internally implemented as a type of the form $id \times ((\,) \rightarrow \tau)$ where id is the type of the identifier. In practice, object identifiers can be implemented in several ways, such as with serial integer numbers or with empty records, and id would be substituted by those types. However, how to implement identifier would not have any effect to other part of this development, therefore, here we denote the type of identifiers just as id.

2.3.2 Examples of views

The mechanism defined above allows us to represent most of the features of views discussed in literature. We demonstrate below some of them.

Basic functionalities of views: The example below includes attribute renaming, attribute hiding, computed attribute and access restriction.

```
joe = obj([Name = "Joe", BirthYear = 1955, Salary := 2000, Bonus := 5000])
    : obj([Name = string, BirthYear = int, Salary := int, Bonus := int])
joe_view = (joe as  λx.[Name = x·Name, Age = ThisYear() - x·BirthYear,
                        Income = x·Salary, Bonus := extract(x, Bonus)])
         : obj([Name = string, Age = int, Income = int, Bonus := int])
```

joe and joe_view are objects with the same identifier, and objeq(joe, joe_view) is true. joe_view renames Salary attribute to Income, hides BirthYear, adds a "computed attribute" Age. It also prohibit the user from updating Salary field by making Income to be immutable.

In some cases, we want to define a view that has more information than the original object. Such a view can be defined by using constants if that information would not be updated, or by using an additional record if that information might be updated.

```
joe_view2 = (joe_view as  let r= [Nickname := "Smile"] in
                          λx.[Name = x·Name, Age = x·Age,
                              Sex = "Male", Nickname := extract(r, Nickname)]
                      end)
          : obj([Name = string, Age = int, Sex = string, Nickname := string])
```

In this view, an immutable field Sex is added using a constant and a mutable field Nickname is added using an additional record. The additional record would be included in the new function closure and carried together with other attribute information. This example also shows how we can accumulate views. We define joe_view2 on top of another view, joe_view, just the same way as we defined joe_view.

Query on objects: The calculus achieves a uniform treatment of views and queries; i.e. they are essentially the same. Queries correspond to evaluation of views using query instead of composition of views using as. For example, if we define a function

$$\begin{aligned}&\mathsf{let\ annualIncome} = \lambda x.x{\cdot}\mathsf{Income} * 12 + x{\cdot}\mathsf{Bonus}\\&\quad : \forall t::[\![\mathsf{Income} = int, \mathsf{Bonus} = int]\!].\ t \rightarrow int\end{aligned}$$

then the expression

$$\mathsf{query(annualIncome,joe\_view)}$$

yields 29000. Moreover, query functions can be arbitrarily complex functions definable in the language.

We are also able to directly define functions for objects instead of functions for records. For example, we can define the following function instead of annualIncome above.

$$\mathsf{let\ annualIncome2} = \lambda o.\mathsf{query}(\lambda x.x{\cdot}\mathsf{Income} * 12 + x{\cdot}\mathsf{Bonus},o)$$

Then, our type system can infer the principle type of this function as follows:

$$\forall t::[\![\mathsf{Income} = int, \mathsf{Bonus} = int]\!].\ obj(t) \rightarrow int$$

As we can see in this example, our type system is properly lifted to object world so that it can work for polymorphic functions for objects just the same way it works for polymorphic record functions.

In some cases, however, we must take care about where we should use query. For example, suppose o is bound to the following object.

$$\begin{aligned}&\mathsf{let\ o} = (\mathsf{obj}([\mathsf{Name} = \text{"Doe"},\ \mathsf{Age} := 24])\\&\qquad\qquad \mathsf{as}\ \lambda x.[\mathsf{Name} = x{\cdot}\mathsf{Name},\ \mathsf{Age} := \mathsf{extract}(x, \mathsf{Age}),\ \mathsf{Class} = x{\cdot}\mathsf{Age\ div\ 5}]);\end{aligned}$$

where div is division for integer. Then the following two programs evaluated to different results.

$$\begin{aligned}&\mathsf{let\ f} = \lambda x.(\mathsf{update}(x, \mathsf{Age}, x{\cdot}\mathsf{Age} + 1)\ ;\ x{\cdot}\mathsf{Class})\ \mathsf{in}\\&\qquad \mathsf{query(f,o)}\\&\mathsf{end}\end{aligned}$$

$$\begin{aligned}&\mathsf{let\ f} = \lambda x.(\mathsf{query}(\lambda y.\mathsf{update}(y, \mathsf{Age}, y{\cdot}\mathsf{Age} + 1),x)\ ;\ \mathsf{query}(\lambda z.z{\cdot}\mathsf{Class},x))\ \mathsf{in}\\&\qquad (\mathsf{f\ o})\\&\mathsf{end}\end{aligned}$$

";" in the programs above means sequential execution that can be easily defined as a macro expression. The former programs returns 4 while the latter returns 5. In some cases, the former one may be appropriate, and in other cases, the latter may be appropriate. Because our calculus can have clear semantics about the evaluation of views by explicitly using query construct, the user can choose either case appropriately.

View update: View update can be done by simply writing a query that changes some of mutable fields of a view by using an ordinary function that update record fields. For example,

```
adjustBonus = λo.query(λx.update(x, Bonus, x·Income * 3),o)
    : ∀t::[Income=int, Bonus:=int].obj(t)→unit
```

is used as a query that adjust an object's Bonus attribute. By applying this query as in

```
(adjustBonus joe_view)
```

Bonus field is correctly updated. After this query, query(λx.x,joe_view) will yield

```
[Name="Joe", Age=39, Income=2000, Bonus:=6000]
```

It should be noted that the renaming of attributes is transparent to programmers; he can program any update allowed by the type of a view. Note also that view evaluation is done lazily, so that an update made through one view is correctly reflected to any other views sharing the same attribute information. So, after this, the query query(λx.x,joe) will yield

```
[Name="Joe", BirthYear=1955, Salary:=2000, Bonus:=6000]
```

where the change of Bonus field through joe_view is correctly reflected.

Manipulation of sets of objects: By combining objects with views of the same view type, we can write a query against a set of heterogeneous objects. The following function returns the set of objects whose annualIncome exceeds 100000.

```
let wealthy = λS.select as λx.[Name=x·Name, Age=x·Age]
                  from S
                  where λx.query(annualIncome,x) > 100000
                  : ∀t1::U.∀t2::U.∀t3:: [Income=int, Bonus=int, Name = t1, Age=t2].
                    {obj(t3)} → {obj([Name=t1,Age=t2])}
```

which can be applied to any set of objects, even if whose internal structure may be heterogeneous, as long as whose view type contains Name, Age, Income, and Bonus fields, as seen in the example:

```
let EmployeeRecords = {...}
    : {[Name = string, BirthYear = int, Salary := int, Bonus := int]}
let ManagerRecords = {...}
    : {[Name = string, Age := int, Income := int, Bonus := int, Section := string]}
let EmployeeObjects = map(λx.(obj(x) as λy.[Name = y·Name, Age = ThisYear() - y·BirthYear,
                                            Income = y·Salary, Bonus = y·Bonus]),
                          EmployeeRecords)
    : {obj([Name = string, Age = int, Income = int, Bonus = int])}
let ManagerObjects = map(λx.(obj(x) as λy.[Name = y·Name, Age = y·Age,
                                            Income = y·Income, Bonus = y·Bonus]),
```

```
                    ManagerRecords)
        : {obj([Name = string, Age = int, Income = int, Bonus = int])}
let S = union(EmployeeObjects, ManagerObjects)
        : {obj([Name = string, Age = int, Income = int, Bonus = int])}
(wealthy S)
        : obj([Name=string, Age=int])
```

Creating a new relation from sets of objects are also be easily done by using relation $\cdots$ from $\cdots$ where $\cdots$ constructs we have defined.

```
relation [Employee = x, Manager = y]
  from e ∈ EmployeeObjects, m ∈ ManagerObjects
  where query(λx.query(λy.(x·Income> y·Income),m),e)
: {[Employee = obj([Name = string, Age = int, Income = int, Bonus = int]),
    Manager = obj([Name = string, Age = int, Income = int, Bonus = int])]}
```

This program returns the relations of an employee object and a manager object such that the employee's income is greater than the manager's income.

2.3.3 Typing and semantics of objects and views

We introduce new constructors for objects with views and a type constructor $obj()$ for them. The typing rules for the new expression constructors are shown in Figure 2.3.

The semantics for these new constructors for objects is given by a systematic translation for these new expression constructors into the core language. Figure 2.4 shows a set of rules that recursively eliminate the new constructors we have introduced in this section. *newid* in these translation rules is a expression that creates a new identifier. Because the implementation of this construct depends on how we implement *id*, here we denote it simply as *newid*. One point to note in those translation rules is that any components e_i of those construct should be evaluated only once because the evaluation of those components may cause some side effect, such as record creation or record update. In such cases, the order of evaluation of those components is also important. These translation is defined so that components of a construct are always evaluated in the order of $e_1, \ldots, e_n$.

The view extension preserves the type soundness. We establish this by showing that the translation $\mathbf{tr}(e)$ preserves typing. Let τ be a type of the extended language. Then we define $tr(\tau)$, *internal representation* of τ, as a type of the core language that is obtained by repeatedly substituting any component type of τ of the form $obj(\tau')$ with a type of the form $id \times (() \rightarrow \tau')$. We also define $tr(\mathcal{K})$ and $tr(\mathcal{A})$, the translation of $\mathcal{K}$ and $\mathcal{A}$, as assignments that is obtained from $\mathcal{K}$ or $\mathcal{A}$ by repeatedly substituting all $obj(\tau)$ appearing in it with the internal representation of theirs. For example, the internal representations of $obj([\text{Name} = string]) \rightarrow string$ is $(id \times (() \rightarrow [\text{Name} = \text{string}])) \rightarrow string$.

$$\text{(obj)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e : \tau}{\mathcal{K},\mathcal{A} \rhd \mathsf{obj}(e) : obj(\tau)}$$

$$\text{(vcomp)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e_1 : obj(\tau_1) \qquad \mathcal{K},\mathcal{A} \rhd e_2 : \tau_1 \rightarrow \tau_2}{\mathcal{K},\mathcal{A} \rhd (e_1 \ \mathsf{as}\ e_2) : obj(\tau_2)}$$

$$\text{(query)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e_1 : \tau_1 \rightarrow \tau_2 \qquad \mathcal{K},\mathcal{A} \rhd e_2 : obj(\tau_1)}{\mathcal{K},\mathcal{A} \rhd \mathsf{query}(e_1,e_2) : \tau_2}$$

$$\text{(fuse)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e_1 : obj(\tau_1) \qquad \mathcal{K},\mathcal{A} \rhd e_2 : obj(\tau_2)}{\mathcal{K},\mathcal{A} \rhd \mathsf{fuse}(e_1,e_2) : \{obj(\tau_1 \times \tau_2)\}}$$

$$\text{(rel)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e_i : obj(\tau_i)\ (1 \leq i \leq n)}{\mathcal{K},\mathcal{A} \rhd \mathsf{relobj}(l_1 = e_1, \ldots, l_n = e_n) : obj([l_1 = \tau_1, \ldots, l_n = \tau_n])}$$

Figure 2.3: Typing Rules for Objects and Views

We then have the following desirable property:

Proposition 3 *Let e be an expression possibly containing object expressions. If $\mathcal{K},\mathcal{A} \rhd e : \tau$ is derivable in the extended language, then $tr(\mathcal{K}), tr(\mathcal{A}) \rhd \mathbf{tr}(e) : tr(\tau)$ is derivable in the core language.*

Proof outline This can be easily proved by the induction on the structures of the expressions of the extended language. ∎

We show some examples. By the typing rules for the extended language the assertion below is derivable.

$$\{\}\{\} \rhd (\mathsf{obj}([\mathsf{Name} = \text{"Joe"}, \mathsf{Age} = 35])\ \mathsf{as}\ \lambda x.[\mathsf{Name} = x{\cdot}\mathsf{Name}])$$
$$: obj([\mathsf{Name} = string])$$

Then the assertion on the translation of this expression and the internal representation of the type above is derivable in the core language as below.

$$\{\}\{\} \rhd ((\lambda o.\lambda f.(o{\cdot}1,\ \lambda().(f\ (o{\cdot}2\ ()))))$$
$$(\lambda o.(newid,\ \lambda().o)\ \ [\mathsf{Name} = \text{"Joe"}, \mathsf{Age} = 35]))$$
$$\lambda x.[\mathsf{Name} = x{\cdot}\mathsf{Name}])$$
$$: id \times (() \rightarrow [\mathsf{Name} = string])$$

Similarly, the following assertion is derivable in the extended language:

$$
\begin{aligned}
\mathbf{tr}(\mathsf{obj}(e)) &= (\lambda o.\langle newid,\ \lambda().o\rangle\ \ \mathbf{tr}(e)) \\
\mathbf{tr}((e_1\ \mathsf{as}\ e_2)) &= ((\lambda o.\lambda f.\langle o{\cdot}1,\ \lambda().(f\ (o{\cdot}2\ ()))\rangle\ \ \mathbf{tr}(e_1))\ \ \mathbf{tr}(e_2)) \\
\mathbf{tr}(\mathsf{query}(e_1,e_2)) &= ((\lambda f.\lambda o.(f\ (o{\cdot}2\ ()))\ \ \mathbf{tr}(e_1))\ \ \mathbf{tr}(e_2)) \\
\mathbf{tr}(\mathsf{fuse}(e_1,e_2)) &= ((\lambda o1.\lambda o2. \\
&\qquad \mathsf{if}\ \mathsf{eq}(o1{\cdot}1, o2{\cdot}1)\ \mathsf{then}\ \{\langle o1{\cdot}1,\ \lambda().(\langle o1{\cdot}2\ (),\ \langle o2{\cdot}2\ ()\rangle\rangle)\rangle\}\ \mathsf{else}\ \{\} \\
&\quad \mathbf{tr}(e_1))\ \ \mathbf{tr}(e_2)) \\
\mathbf{tr}(\mathsf{relobj}(l_1 = e_1, \ldots, l_n = e_n)) &= (\cdots(\lambda o1.\cdots\lambda on. \\
&\qquad \langle newid,\ \lambda().[l_1{=}(o1{\cdot}2\ ()), \ldots,\ l_n{=}(on{\cdot}2\ ())]\rangle \\
&\quad \mathbf{tr}(e_1))\cdots\ \ \mathbf{tr}(e_n))
\end{aligned}
$$

Figure 2.4: Semantics of Objects and Views through Transformation

$\{t{::}[\![\mathsf{Supervisor} = obj([\mathsf{Name} = string])]\!]\}$
$\{\mathsf{f}{:}obj(t){\to}obj([\mathsf{Name} = string]),\ \mathsf{o}{:}obj([\mathsf{Name} = string,\ \mathsf{Supervisor} = obj([\mathsf{Name} = string])])\}$
$\quad \triangleright(\mathsf{f}\ \mathsf{o}) : obj([\mathsf{Name} = string])$

Corresponding to it, the assertion below is derivable in the core language:

$\{t{::}[\![\mathsf{Supervisor} = id \times (()\to[\mathsf{Name} = string])]\!]\}$
$\{\mathsf{f}{:}(id \times (()\to t))\to(id \times (()\to[\mathsf{Name} = string])),$
$\ \mathsf{o}{:}id \times (()\to[\mathsf{Name} = string,\ \mathsf{Supervisor} = id \times (()\to[\mathsf{Name} = string])])\}$
$\ \triangleright(\mathsf{f}\ \mathsf{o}) : id \times (()\to[\mathsf{Name} = string])$

This proposition and the soundness result of the core language (Proposition 1) imply that the polymorphic type system with objects and views is sound with respect to the operational semantics defined by translating the extended language with the function $\mathbf{tr}(\,)$ and then evaluating the resulting expression in the core language. Therefore, the slogan that "a well typed program cannot go wrong" also hold for our language with objects and views. The extended language also preserves the existence of a complete type inference algorithm. This is easily seen from the shape of the new typing rules. Any *tau* appearing in the bottom of the rules for the constructs for objects also appears in the top of the rule as a part of a result of typing of component of the construct. Therefore, if we compute the principal type of those subexpressions, we can also compute the principal type of the object expressions.

2.4 Classes and Object Sharing

We are now going to develop a mechanism for classes and object sharing among them on top of the language we have just defined. Classes in this calculus are not used as templates for object creation nor as method dictionaries but they are used to organize objects in a databases. (However, classes with such features also can be implemented on top of this calculus as described in a later section.) Classes have their extent, which is a set of objects of uniform view type but possibly of heterogeneous internal structure. The extent of a class consists of two components; its own extent which is a set of objects that directly belong to that class, and its imported extent which is a set of object imported from other classes. The own extent is statically defined by enumeration, while the imported extent is defined by predicates and is dynamically computed. Therefore, classes can be regarded as a special kind of sets that are defined with both intension and extension.

In usual object-oriented databases, the extent of a class similarly consists of two components, its own extent and the extent inherited from its subclasses. As explained in the introduction, this model can express only limited form of object sharing. Some researches have introduced classes which is defined by giving how the extent of that class is dynamically derived from other classes' extent. Introducing those classes that exist only virtually without their physical substance, however, cause some problems. Those virtual classes and usual classes cannot be uniformly manipulated, especially when they are updated. In our model, every class has same status and share instances with each other on a equal term. Therefore, no distinction would be needed. Even updates to each classes can be bidirectionally propagated by using mutually recursive class definition we explain later in this section. Moreover, introducing those virtual classes in addition to the usual inheritance mechanism seems to be ad-hoc way to make up for a deficiency of usual inheritance model. Our approach, which allows users to define any object sharing relation between arbitrary classes, is more natural way. Actually, it successfully provide a highly general and flexible framework as described in this section.

2.4.1 Class definitions and their typing

To deal with classes, the syntax of the language is extended with the following constructors:

$$
\begin{aligned}
e ::= \cdots \mid\ & \mathsf{class}\ e\ \mathsf{include}\ e, \ldots, e\ \mathsf{as}\ e\ \mathsf{where}\ e \\
& \qquad \vdots \\
& \qquad \mathsf{include}\ e, \ldots, e\ \mathsf{as}\ e\ \mathsf{where}\ e\ \mathsf{end} \\
\mid\ & \mathsf{c\text{-}query}(e, e) \mid \mathsf{insert}(e, e) \mid \mathsf{delete}(e, e)
\end{aligned}
$$

The intended meanings of these constructors are explained below. Their precise semantics is given later by defining a translation of them into the language for objects.

Class definition: $\mathsf{class}\ S\ \mathsf{include}\ C_1^1, \ldots, C_1^{m_1}\ \mathsf{as}\ e_1\ \mathsf{where}\ p_1 \ldots \mathsf{include}\ C_n^1, \ldots, C_n^{m_n}\ \mathsf{as}\ e_n\ \mathsf{where}\ p_n\ \mathsf{end}$

The first component creates a class, where S is the own extent of the class, and each include $C_1, \ldots, C_n$ as e where p clause asserts that the class being defined includes all the objects that satisfy p from the intersection (in the sense of intersect) of the classes $C_1, \ldots, C_n$ under a new view specified by the function e. As seen in the semantic definition below, the actual inclusion does not take place until some query requiring the entire extent of the class is evaluated.

This general form of class definition combines four basic functionalities of object sharing: union through multiple include clauses, intersection, selection (where), and view composition (as). Although we believe that this syntax is general enough to represent most cases of object sharing, some other operations may also be included, and we leave this issue as a future investigation. Here we only point out one issue in selecting operations in class definitions. In analogy with SQL, one might think that the above class definition would be more general if one would have interpreted the set of classes $C_1, \ldots, C_n$ in an include clause as product formation. However, in our model as well as most of object-oriented data models, forming a product or more generally forming a record implies creation of new identity. In our model, a class definition involving an identity creating operation cannot be given a well founded semantics. This is a natural consequence of the intended semantics of a class definitions; in our model a class specifies new views and sharing of existing objects from other classes, both of them do not imply creation of new object identity.

Class query: c-query(e_1, e_2).

This evaluates a query specified by a function e_1 against a class e_2, where e_1 may be any function on sets of objects whose type is the same as that of objects of the class. This construct allows the programmer to treat classes just as sets of objects. Intuitively, this construct first compute the extent of the class e_2 by evaluating its inclusion predicates and apply e_1 to the computed extent. It is only this construct that materialize the extent of classes.

Insertion: insert(e_1, e_2).

This inserts an object e_1 to a class e_2's own extent. If some other class is including objects from the class e_2, and object e_1 satisfies the inclusion predicates, then this insertion should be also reflected to that class. As explained later, this update propagation is realized by dynamically creating the extent of classes immediately before evaluating queries on classes.

Deletion: delete(e_1, e_2).

This removes an object e_1 from a class e_2's own extent. In the same way as insertion, this removal is propagated to classes including objects from this class. Other semantics for delete can also be possible. We could define it so that if the specified element is imported from another class then it removes the element from that class, or it blocks the inclusion of the element from that class. The rationale of our choice is clarity and safety.

The extended syntax allows any mixture of classes and other expression constructors, i.e. classes can be treated as first-class values. This opens up the possibility of various powerful programming styles with classes, such as using class creating functions.

$$\text{(class)}\quad \frac{\mathcal{K},\mathcal{A} \rhd S : \{obj(\tau)\} \qquad \mathcal{K},\mathcal{A} \rhd e_i : \tau_i^1 \times \cdots \times \tau_i^{m_i} \to \tau \qquad \mathcal{K},\mathcal{A} \rhd C_i^j : class(\tau_i^j) \qquad \mathcal{K},\mathcal{A} \rhd p_i : obj(\tau_i^1 \times \cdots \times \tau_i^{m_i}) \to bool}{\mathcal{K},\mathcal{A} \rhd \mathsf{class}\ S \mathsf{include} \cdots \mathsf{as}\ e_1 \mathsf{where}\ p_1 \cdots \mathsf{include} \cdots \mathsf{as}\ e_n \mathsf{where}\ p_n : class(\tau)}$$

$$\text{(cquery)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e_1 : \{obj(\tau_1)\} \to \tau_2 \qquad \mathcal{K},\mathcal{A} \rhd e_2 : class(\tau_1)}{\mathcal{K},\mathcal{A} \rhd \mathsf{c\text{-}query}(e_1, e_2) : \tau_2}$$

$$\text{(insert)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e_1 : class(\tau_1) \qquad \mathcal{K},\mathcal{A} \rhd e_2 : obj(\tau_1)}{\mathcal{K},\mathcal{A} \rhd \mathsf{insert}(e_1, e_2) : unit}$$

$$\text{(delete)}\quad \frac{\mathcal{K},\mathcal{A} \rhd e_1 : class(\tau_1) \qquad \mathcal{K},\mathcal{A} \rhd e_2 : obj(\tau_1)}{\mathcal{K},\mathcal{A} \rhd \mathsf{delete}(e_1, e_2) : unit}$$

Figure 2.5: Typing Rules for Classes

To define typing rules for classes, we introduce a new type constructor for classes:

$$\tau ::= \cdots \mid class(\tau)$$

denoting a class of objects having type $obj(\tau)$. The typing rules for classes and operations on classes are given in Figure 2.5.

2.4.2 An example of a class

We show a simple example of a class. Suppose Staff and Student are classes already defined with type $class([\mathsf{Name} = string, \mathsf{Age} = int, \mathsf{Sex} = string, \mathsf{Salary} := string])$ and $class([\mathsf{Name} = string, \mathsf{Age} = int, \mathsf{Sex} = string, \mathsf{Degree} := int])$. The following example defines a new class FemaleMember:

```
let FemaleMember = class {}
      includes Staff
        as λs.[Name = s·Name, Age = s·Age, Category = "staff"]
        where λs.query(λx.eq(x·Sex,"female"),s)
      includes Student
        as λs.[Name = s·Name, Age = s·Age, Category = "student"]
        where λs.query(λx.eq(x·Sex,"female"),s)
  : class([Name = string, Age = int, Category = string])
```

In this definition, the initial value of the own extent of FemaleMember is the empty set. It can be updated after the class formation by using insert and delete operations. Class FemaleMember shares those objects

of Staff and Student whose Sex is "female". Salary field of Staff objects, and Degree field of Student objects, and Sex field of both of them are hidden, and an additional field Category is set appropriately.

Below is an example of a query against the class FemaleMember:

$$\begin{array}{l}\mathsf{let\ names} = \lambda \mathsf{s.map}(\lambda \mathsf{x.query}(\lambda \mathsf{y.y{\cdot}Name,x}),\ \mathsf{s}) \\ \quad : \forall t_1 :: U.\forall t_2 :: [\![\mathsf{Name} = t_1]\!].\ \{obj(t_2)\} \to \{t_1\} \\ \mathsf{c\text{-}query(names,\ FemaleMember)} \\ \quad : \{string\}\end{array}$$

The function names is the operation on usual sets of objects. We can issue a query on a class using usual set operations and c-query construct. The extent of FemaleMember is computed just before this c-query is evaluated, so the result of this query, the names of all members that are considered as female, reflects the latest status of Student and Staff.

The following is an example of class definition where an include clause contains multiple classes:

$$\begin{array}{l}\mathsf{StudentStaff} = \mathsf{class}\ \{\} \\ \quad \mathsf{includes\ Staff,\ Student} \\ \quad\quad \mathsf{as}\ \lambda \mathsf{p.[Name} = \mathsf{p{\cdot}1{\cdot}Name,\ Age} = \mathsf{p{\cdot}1{\cdot}Age,\ Sex} = \mathsf{p{\cdot}1{\cdot}Sex,} \\ \quad\quad\quad\quad \mathsf{Sal} := \mathsf{extract(p{\cdot}1, Salary),\ Deg} := \mathsf{extract(p{\cdot}2, Degree)]} \\ \quad\quad \mathsf{where}\ \lambda \mathsf{p.true} \\ \quad : \mathit{class}([\mathsf{Name} = \mathit{string},\ \mathsf{Age} = \mathit{int},\ \mathsf{Sex} = \mathit{string},\ \mathsf{Sal} := \mathit{string},\ \mathsf{Deg} := \mathit{int}])\end{array}$$

Intuitively StudentStaff corresponds to the intersection class of Staff and Student.

2.4.3 Semantics of classes

Classes are sets of objects that are evaluated lazily so that updates to classes propagate properly through sharing predicates. The precise semantics of classes is defined by giving a systematic translation of classes into the extended core language defined in the previous section. This also provides an effective algorithm to implement the extended language with classes.

The translation **tr**( ) for expressions including classes is given in Figure 2.6. As seen in these definitions, lambda abstraction, $\lambda().\cdots$, of inclusion functions *delays* the materialization of the extent inclusion from other classes. The actual extent is created immediately before a query in c-query expression. The translation of type constructor *class* is defined as:

$$tr(\mathit{class}(\tau)) = [\mathsf{OwnExt}{:=}\{obj(\tau)\},\ \mathsf{Ext}{:=}\ \mathit{unit} \to \{obj(\tau)\}]$$

As seen in the translation above, function type is used only for delaying the extent creation and therefore its input type is the trivial type *unit*. The translation for kind assignment or type assignment, denoted as $tr(\mathcal{K})$ or $tr(\mathcal{A})$, is also defined as the assignment that is obtained by repeatedly substituting all types appearing in it of the form $\mathit{class}(\tau)$ with $tr(\mathit{class}(\tau))$.

$$
\begin{array}{lcl}
\mathbf{tr}(\text{class } S \text{ include } C_1^1,\ldots,C_1^{m_1} \text{ as } e_1 \text{ where } p_1 \cdots \text{include } C_n^1,\ldots,C_1^{m_n} \text{ as } e_n \text{ where } p_n \text{ end}) & & \\
\quad = & [\text{OwnExt} := S, & \\
& \text{Ext} := \lambda().\text{union}(S, \text{union}(\ldots, \text{union}(& \\
& \quad (\text{select as } \mathbf{tr}(e_i) \text{ from intersect}(\ldots, (\mathbf{tr}(C_i^j)\cdot\text{Ext } ()), \ldots) \text{ where } \mathbf{tr}(p_i)), & \\
& \quad \text{union}(\cdots))))] & \\
\mathbf{tr}(\text{c-query}(e, C)) & = & (\mathbf{tr}(e)\ (\mathbf{tr}(C)\cdot\text{Ext } ())) \\
\mathbf{tr}(\text{insert}(e, C)) & = & (\lambda\text{c.update}(\text{c}, \text{OwnExt}, \text{union}(\{\mathbf{tr}(e)\}, \text{c-OwnExt}))\ \mathbf{tr}(C)) \\
\mathbf{tr}(\text{delete}(e, C)) & = & (\lambda\text{c.update}(\text{c}, \text{OwnExt}, \text{remove}(\mathbf{tr}(e), \text{c-OwnExt}))\ \mathbf{tr}(C))
\end{array}
$$

Figure 2.6: Semantics of Classes through Transformation

We can then prove the following property:

Proposition 4 *Let e be an expression possibly containing classes and operations on classes. If $\mathcal{K}, \mathcal{A} \rhd e : \tau$ is derivable in the extended language then $tr(\mathcal{K}), tr(\mathcal{A}) \rhd \mathbf{tr}(e) : tr(\tau)$ is derived in the language with objects defined in Section 3.*

Proof outline This can be easily proved by the induction on the structures of the expressions of the extended language. ∎

This property together with Proposition 1 and 3 establishes the soundness of the language with classes. As in the case of view extension, it is easily proved that the class extension preserves the existence of a complete type inference algorithm.

2.4.4 Recursive class definition

The above typing rule for class definition and its semantics do not allow cyclic sharing among various classes, which is sometimes useful. To support cyclic sharing, we add the following recursive class definition:

$$
\begin{array}{l}
\text{let } c_1 = \text{class } S_1 \\
\quad \text{include } {}_1C_1^1, \ldots {}_{l_1^1}C_1^1 \text{ as } e_1^1 \text{ where } p_1^1 \cdots \\
\quad \text{include } {}_1C_1^{m_1}, \ldots {}_{l_1^{m_1}}C_1^{m_1} \text{ as } e_1^{m_1} \text{ where } p_1^{m_1} \\
\text{and } c_2 = \cdots \\
\vdots \\
\text{and } c_n = \text{class } S_n \\
\quad \text{include } {}_1C_n^1, \ldots {}_{l_n^1}C_n^1 \text{ as } e_n^1 \text{ where } p_n^1 \cdots \\
\quad \text{include } {}_1C_n^{m_n}, \ldots {}_{l_n^{m_n}}C_n^{m_n} \text{ as } e_n^{m_n} \text{ where } p_n^{m_n} \\
\text{in } e \text{ end}
\end{array}
$$

$$\text{(rec-class)}\quad \frac{\begin{array}{l}\mathcal{K},\mathcal{A} \rhd S_i : \{obj(\tau_i)\} \\ \mathcal{K},\mathcal{A}\{c_1 : class(\tau_1),\ldots,c_n : class(\tau_n)\} \rhd_k C_i^j : class({}_k\tau_i^j) \\ \mathcal{K},\mathcal{A} \rhd e_i^j : {}_1\tau_i^j \times \cdots \times_{l_i^j} \tau_i^j \to \tau_i \\ \mathcal{K},\mathcal{A} \rhd p_i^j : obj({}_1\tau_i^j \times \cdots \times_{l_i^j} \tau_i^j) \to bool \\ \mathcal{K},\mathcal{A}\{c_1 : class(\tau_1),\ldots,c_n : class(\tau_n)\} \rhd e : \tau\end{array}}{\mathcal{K},\mathcal{A} \rhd \mathsf{let}\ c_1 = \cdots \ \mathsf{and} \cdots \mathsf{and}\ c_n = \cdots \ \mathsf{in}\ e\ \mathsf{end} : \tau}$$

Figure 2.7: Typing rule for recursive class definition

${}_kC_i^j$ are either one of class identifiers $c_1,\ldots,c_n$ or any class expression that does not contain any of $c_1,\ldots,c_n$. e_i and p_i are as before and cannot contain class identifiers $c_1,\ldots,c_n$. Therefore, $c_1,\ldots,c_n$ can appear recursively only in a form of "include ...,c_i,...". This restricted use of recursion enables us to define a well founded semantics for the recursive class definitions.

The typing rule for this recursive class definition is given in Figure 2.7. Note that the restriction that "c_i cannot appear in p_i^j and e_i^j" is correctly enforced by using the type assignment $\mathcal{A}$ for p_i^j and e_i^j that does not contain the assumption on the variables $c_1,\ldots,c_n$. Although the another restriction, "$c_1,\ldots,c_n$ can not be a component of some expression even in include clauses", is not expressed in the typing rules, it also can be statically checked with ease. We show some examples to explain those restrictions:

Suppose we attempts to define a class C and two mutually recursive classes C_1 and C_2 to represent the relation $C_1 = C \setminus C_2$ and $C_2 = C \setminus C_1$ by the code:

```
let C = class {o}
in let  C1 = class {} includes C as ··· where λx.c-query(λy.not(member(x,y)), C2)
   and C2 = class {} includes C as ··· where λx.c-query(λy.not(member(x,y)), C1)
   in ··· end
end
```

However, the intended semantics is itself ambiguous. Either "$C_1 = \{o\}$ and $C_2 = \{\}$" or "$C_1 = \{\}$ and $C_2 = \{o\}$" can satisfy the intended sharing relation. In fact, this code cannot be well typed by our typing rules. On the other hand, the following recursive definition has clear meaning, and be well typed in our typing rules.

```
let C = class {o1, o2}
in let  C1 = class {o1, o3} includes C2 as ··· where λx.c-query(λy.not(member(x,y)), C)
   and C2 = class {o2, o4} includes C1 as ··· where λx.c-query(λy.not(member(x,y)), C)
   in ··· end
```

```
end
```

Intuitively, this definition intends the sharing relation $C_1 \supset C_2 \setminus C$ and $C_2 \supset C_1 \setminus C$. The appropriate solution is $C_1 = \{o_1, o_3, o_4\}$ and $C_2 = \{o_2, o_3, o_4\}$.

Using c_i as an expression component in the include clause also causes ambiguous definitions. For example:

```
let C = class {o}
in let  C1 = class {} includes setminus(C, C2) as ··· where λx.true
   and C2 = class {} includes setminus(C, C2) as ··· where λx.true
   in ··· end
end
```

Here, assume that setminus(C, C') is a function that has a type $\forall t::U.class(t) \times class(t') \rightarrow class(t)$ and returns a class with a extent $C \setminus C'$. This definition causes ambiguity in the same sense as the example above.

As mentioned earlier, we limit the operations in include clause so that they does not include any identity creation. If they does, for example, if we choose Cartesian product in the place of intersection, infinite identity creation may occur. Suppose the following example:

```
let C = {o}
in let  C1 = class {o1} include C2 as ··· where λx.true
   and C2 = class {} include C1, C as ··· where λx.true
   in ··· end
end
```

Then, the computation of the extent of C_1 becomes infinite. The extent of C_1 must includes o_1, (o_1, o), $((o_1, o), o)$, $(((o_1, o), o), o)$, $\cdots$.

One example that may be sometime useful but that cannot be dealt with in our framework is the definition of "closure" class. For example:

```
let Person = {p1, p2, p3, ...}
let fathers = λS.map(λy.query(λz.z·Father,y), S)
let MyAncestor = {me} include Person as ···
                        where λx.member(x, c-query(fathers, MyAncestor))
```

This class intuitively means the closure produced by me object through the Father field. But such class definition is not supported in our framework. Indeed, this recursive use of MyAncestor cause type error in our typing rules.

Let us show examples of classes with useful cyclic sharing. In the previous example on FemaleMember, if some object is newly inserted to the own extent of FemaleMember, we would like to make that object

shared appropriately by either of Staff or Student. This implies FemaleMember and the other two classes should perform mutual sharing. Then we can use the following recursive definition.

```
let  Staff = class {}
       includes FemaleMember
         as λf.[Name = f·Name, Age = f·Age, Sex = "female"]
         where λf.query(λx.(x·Category = "staff"),f)
and Student = class {}
       includes FemaleMember
         as λf.[Name = f·Name, Age = f·Age, Sex = "female"]
         where λf.query(λx.(x·Category = "student"),f)
and FemaleMember = class {}
       includes Staff
         as λs.[Name = s·Name, Age = s·Age, Category = "staff"]
         where λs.query(λx.(x·Sex = "female"),s)
       includes Student
         as λs.[Name = s·Name, Age = s·Age, Category = "student"]
         where λs.query(λx.(x·Sex = "female"),s)
in ··· end
```

By this definition, update to each classes is appropriately propagated to the other classes. When some user seeing class Staff or Student inserts into them a new instance which is also a female, that insertion will be appropriately propagated to FemaleMember, and when some user seeing class FemaleMember inserts to it a new instance which is also a staff or a student, that insertion will be appropriately propagated to Staff or Student. Similarly, class StudentStaff can be defined with the following cyclic sharing.

```
let  Staff = class {}
       include StudentStaff
         as λs.[Name = s·Name, Age = s·Age, Sex = s·Sex, Salary := extract(s, Sal)]
         where λs.true
and Student = class {}
       include StudentStaff
         as λs.[Name = s·Name, Age = s·Age, Sex = s·Sex, Degree := extract(s, Deg)]
         where λs.true
and StudentStaff = class {}
       includes Staff, Student
         as λp.[Name = p·1·Name, Age = p·1·Age, Sex = p·1·Sex,
                 Sal := extract(p·1, Salary), Deg := extract(p·2, Degree)]
         where λp.true
```

Figure 2.8: The directed acyclic graph illustrating total inclusion relation

One may consider that one of the roles of IS-A hierarchy is to represent the hierarchical classification of objects to the user. The hierarchical classification is very easy to understand and helps the user to grasp the organization of objects. However, providing the users with easily understandable hierarchical classification and controlling the object sharing among classes are different issues. Indeed, even in our framework, it is not difficult for the system to detect the all include clauses using true in its where part, and to display a directed acyclic graph illustrating which class totally includes which class. For example, from the definitions above, the system can construct and display the directed acyclic graphs shown in Figure 2.8.

Intuitively, the semantics of recursive class definitions can be understood as follows. First, recursive use of the name of the classes can appear only in a form like:

let $c_1 = \cdots$ include $\ldots, c_2, \ldots$ as $\cdots$
and $c_2 = \cdots$ include $\ldots, c_3, \ldots$ as $\cdots$
and $c_3 = \cdots$ include $\ldots, c_1, \ldots$ as $\cdots$

Then, the computation of the extent of c_1 needs the computation of the extent of c_2, and that computation, in turn, needs c_3, and then c_1 is needed again. At that time, however, the computation of c_1 is not really necessary. The objects to be included through include $\ldots, c_1, \ldots$ is intersection of those classes, therefore, those objects are already included in c_1. Since we have defined that set of objects is collapsed based on objeq, those objects need not be included again.

Then, we formally define an effective semantics for the recursive class definition. Some care must be taken so that repeated inclusion of objects is avoided. Let C be a recursive class definition of the form:

let $c_1 = \cdots$
and $c_i =$ class S_i
include ${}_1C_i^1, \ldots, {}_{l_i^1}C_i^1$ as e_i^1 where $p_i^1 \cdots$
include ${}_1C_i^{m_i}, \ldots, {}_{l_i^{m_i}}C_i^{m_i}$ as $e_i^{m_i}$ where $p_i^{m_i}$
$\vdots$
and $c_n = \cdots$
in e end

We first define the set of mutually recursive function $\{f^i | 1 \leq i \leq n\}$ corresponding to the set of class identifiers $\{c_1, \ldots, c_n\}$ as follows:

$$f^i = \lambda L.\lambda().\ S_i \cup \bigcup_{1 \le j \le m_i} (\textsf{select as } e_i^j \textsf{ from intersect}(\overline{{}_1C_i^j}, \ldots, \overline{{}_{l_i^j}C_i^j}) \textsf{ where } p_i^j)$$

where $\overline{{}_kC_i^j}$ is the following expression:

$$\begin{array}{l}\textsf{if } {}_kC_i^j = c_a \textsf{ then} \\ \quad \textsf{if } a \in L \textsf{ then } \{\} \textsf{ else } ((f^a\ \ (L \cup \{a\}))\ ()) \\ \textsf{else } \mathbf{tr}({}_kC_i^j)\end{array}$$

The application $((f^i\ \ \{i\})\ ())$ computes the extent denoted by the class identifier c_i. The extra parameter L to f^i indicates the set of functions $\{f^a | a \in L\}$ that indirectly invoke the application in question. Thus if f^i is called with $\{a, \ldots\}$ then this call is to calculate the objects that are included in the class c_a and therefore the call to f^a is not needed.

The recursive definition is well defined in the following sense:

Proposition 5 *There is no infinite calling sequence of* $\{f^1, \ldots, f^n\}$ *starting from any* $(f^i\ \ \{i\})$.

Proof sketch Suppose we have an infinite sequence $(f^{a_1}\ \ L_1), (f^{a_2}\ \ L_2), \ldots, (f^{a_j}\ \ L_j), (f^{a_{j+1}}\ \ L_{j+1}), \ldots$ such that evaluation of $(f^{a_j}\ \ L_j)$ involves evaluation of $(f^{a_{j+1}}\ \ L_{j+1})$. By the definition of each f^i, the following properties can be verified: (1) $|L_{j+1}| = |L_j| + 1$, and (2) $L_j \subseteq \{1, \ldots, n\}$. These two properties contradicts the fact that the sequence is infinite. ∎

This means that extent computation terminates whenever each p_i^j and e_i^j terminates.

This function is routine to translate the above definition into a term in the language with views. The translation for recursive class is then defined as follows:

$$\begin{array}{l}\mathbf{tr}(\textsf{let } \cdots) = \textsf{let } \ c_1 = [\textsf{OwnExt:=}S_1, \textsf{ Ext:=}(f^1\ \ \{1\})] \\ \qquad\qquad \textsf{and } \cdots \\ \qquad\qquad \textsf{and } c_n = [\textsf{OwnExt:=}S_n, \textsf{ Ext:=}(f^n\ \ \{n\})] \\ \qquad\qquad \textsf{in } e \textsf{ end}\end{array}$$

This operational semantics corresponds to the least (with respect to set inclusion) solution to the class definition when we consider it as a system of equation over sets of objects under **objeq**.

2.5 Application

In the previous sections, we provide the highly general basis for object-oriented database languages. They are, indeed, just the formal basis, so we need not necessarily use it directly. Instead, we can define more customized language on top of our calculus. Our calculus is so general that we can flexibly define a wide range of object-oriented features, such as object encapsulation, object creation by classes, and so on. In this section, we show some examples of such applications of our calculus.

2.5.1 Objects as Abstract Data Type Objects

In our calculus, objects are manipulated as if they are usual values, in most case, records. In some applications, however, objects should be abstract data type objects that can be manipulated only limited abstract primitives. We can implement such objects as objects whose view type is the record of those primitive functions. Because attribute information can be directly manipulated only within a state function, encapsulation of implementation of abstract data types can be achieved. For example, an object representing "moving point on one dimension" can be implemented using the following structure:

```
let mp = (obj([x := 10, vx := 2])
          as λp.[Accelerate = λ().λax.update(p, vx, p·vx + ax),
                 Position = λ().p·x,
                 Tick = λ().update(p, x, p·x + p·vx)]);
```

A moving point has its position x and its velocity vx. Primitives to manipulate a moving point is Accelerate which augment the velocity of the point, Position which returns the current position of the point, and Tick which updates the position of the point in accordance with its velocity. () at the head of those primitives is used to prevent that all functions are evaluated every time when this view is materialized. We can manipulate this object only by evaluating one of those functions in its view like below:

```
query(λx.(x·Position ()),mp);
```

Of course, we can define some macros so that we can write those programs more concisely. For example, like below:

```
let mp = (obj([x := 10, vx := 2]) abstract
          λp.[Accelerate = λax.update(p, vx, p·vx + ax),
              Position = p·x,
              Tick = update(p, x, p·x + p·vx)]);

mp <- Position;
```

2.5.2 Classes as Template and Interface of their Instances

While classes in our calculus work only as a container of objects to organize objects in the databases, classes in many object-oriented systems also work as a template for object creation. We can implement classes with that feature on top of our calculus by defining the following macro for definition of such classes.

Class with Object Template: class C: $[l_1 = t_1, \ldots, l_n = t_n]$ as f include ....

In this definition, C is the name of this class, $[l_1 = t_1, \ldots, l_n = t_n]$ is the internal structure of attribute information of objects created by this class, and f is the default viewing function of those objects. This definition is translated to the definition below.

let C = class {} include ...;
let new$C$ = λx1...λxn.let o = (obj([l_1 = x1, ..., l_n = xn]) as f) in insert(o, C);o end;

Combining abstract data type objects and object creating classes, we can implement classes as interface of their instances, i.e. classes defined with fixed instance methods only through which objects can be manipulated. To define such classes, we define the following macro:

Class with Instance Method: class C: [$l_1 = t_1, \ldots, l_n = t_n$] abstract [$m_1 = f_1, \ldots, m_n = f_n$] include ....

For example, we can define class MovingPoint as below:

```
class MovingPoint: [x := int, vx := int] abstract
                   λp.[Accelerate = λax.update(p, vx, p·vx + ax),
                       Position = p·x,
                       Tick = update(p, x, p·x + p·vx)]);
```

Then this definition is translated as below:

```
let MovingPoint = class {};
let newMovingPoint = λx.λvx.let o = ([x := int, vx := int]
                                     as λp.[Accelerate = λ().λax.update(p, vx, p·vx + ax),
                                            Position = λ().p·x,
                                            Tick = λ().update(p, x, p·x + p·vx)])
                                in insert(o, MovingPoint);o end;
```

2.5.3 Incremental Construction of the Schema

In our calculus, we must define classes with all its sharing relations using one class construct. Especially, when we define cyclic sharing, we must define all classes in that cycle in one let ... and ... and ... construct at once. In practice, however, we often need to define the schema of a database in a incremental way. To define classes in the schema in such a way, we introduce the following construct.

Incremental Sharing Definition: C include ... as ... where ....

This construct define a new sharing relation for some already defined class C. When a new sharing relation is defined by this construct, the system examine whether it cause any invalid recursive definition or not. If it does, the system rejects that definition and reports an error. Otherwise, the system redefine the sharing predicate of the class C. Because classes are internally implemented as mutable records and any class importing objects from class C has reference to that record, we can realize this construct by simply updating a mutable field Ext of class C.

We also can introduce some constructs for frequently used kind of class definitions, such as union, intersection or selection class definitions.

Union Class Definition: class C union $c_1, \ldots, c_n$.

This construct defines the class C as the union class of $c_1, \ldots, c_n$. The view type of C is the record including fields common to all of $c_1, \ldots, c_n$, and the data structure for attribute information of C's instance is the record with fields that are used in c_1 to implement those common fields. The below is an example of the definition of a union class.

```
class Adult : [Name = string, BirthYear = int, Children := {Child}]
              as λx.[Name = x·Name, Age = ThisYear() − x·BirthYear, Children := extract(x, Children)];
class Child : [Name = string, Age = int, Father = Adult, Mother = Adult]
              as λx.[Name = x·Name, Age = x·Age, Father = x·Father, Mother = x·Mother];
class Person union Adult and Child;
```

Then, the definition of Person is translated to the program below:

```
class Person : [Name = string, BirthYear = int]
               as λx.[Name = x·Name, Age = ThisYear() − x·BirthYear];
        include Adult as λx.[Name = x·Name, Age = x·Age]
                  where λx.true;
        include Child as λx.[Name = x·Name, Age = x·Age]
                  where λx.true;
```

Intersection Class Definition: class C intersect $c_1, \ldots, c_n$.

This construct defines the class C as the intersection class of $c_1, \ldots, c_n$. The view type of C is the record including all fields of $c_1, \ldots, c_n$, and the structure of attribute information C is the same as that of c_1. We show an example of a intersection class below.

```
class Staff : [Name = string, Sal := int]
              as λx.[Name = x·Name, Salary := extract(x, Sal)];
class Student : [Name = string, Deg := int]
                as λx.[Name = x·Name, Degree := extract(x, Sal)];
class StudentStaff intersect Staff and Student;
```

Then, the last definition of StudentStaff is translated into one class definition and two include constructs as below:

```
class StudentStaff : [Name = string, Sal := int, Deg := int]
                     as λx.[Name = x·Name, Salary := extract(x, Sal) Degree := extract(x, Deg)]
      include Staff, Student
        as λx.[Name = x·1·Name, Salary := extract(x·1, Sal), Degree := extract(x·1, Deg)]
        where λx.true;
class Staff include StudentStaff
            as λx.[Name = x·Name, Salary := extract(x, Sal)]
```

```
                where λx.true;
class Student include StudentStaff
                    as λx.[Name = x·Name, Degree := extract(x, Deg)]
                    where λx.true;
```

Selection Class Definition: class C select c_1 where p

This construct defined the class C as the selection class from c_1 through the predicate p. The view type and the structure of attribute information of C is the same as c_1. For example, if the class Staff is already defined as above, we can define HighSalaryStaff as follows:

```
class HighSalaryStaff select Staff where λx.query(λy.y·Sal,x) > 100;
```

This definition is translated to the definition below:

```
class HighSalaryStaff : [Name = string, Sal := int]
                        as λx.[Name = x·Name, Salary := extract(x, Sal)]
        include Staff as λx.x where λx.query(λy.y·Sal,x) > 100;
class Staff include HighSalaryStaff as λx.x where λx.true;
```

2.6 Contribution and Future Work

We have presented a typed polymorphic functional calculus that supports powerful view definitions and flexible object sharing. In this calculus, one can associate objects with views. Views can be any type consistent functions definable in the calculus. Those objects can be manipulated by ordinary functions for their view type. The necessary code for functions to interface with objects is automatically generated by the system. One can define classes of those objects with general sharing specifications. Each such specification consists of a predicate on the objects to be included and a viewing function, both of which are defined in the same language for manipulating objects. It is also possible to define recursive classes, where inclusion relation contains cycles. We obtained the calculus by successively defining the necessary structures and operations for manipulating views and classes on top of the core calculus for records and sets. We also gave their precise semantics and an effective algorithm to implement them by developing a systematic method to translate those structures and operations into the calculus for records and sets.

The most important contribution of this research is that we provided highly general basis for method sharing and object sharing in object-oriented database models. We believe our approach combining method sharing via polymorphic type inference and object sharing via general predicates can serve as a better alternative and can replace the ordinary method and object sharing via user-defined class hierarchy.

One notable advantage of our approach is that our language uniformly integrates object-oriented database programming and typed polymorphic programming. We believe that this integration will open

up the possibility of transferring various recent results in type theory to object-oriented database programming. For example, it is not hard to integrate the class structures we have developed with parametric classes for object-oriented programming [OB89].

Another interesting issue is abstract characterization of views. In developing a systematic translation for a language with views and classes into a language without them, we noticed an intriguing similarity between certain operations associated with Monad in category theory and our operations for lifting operations for usual values to objects and classes. This suggests that we may be able to find a categorical characterization of object-oriented databases that can be applicable to wider range of data structures. Such an abstract analysis might be useful for comparing expressive powers of languages with views in different paradigms.

Chapter 3

Static Detection of Security Flaws Caused by Encapsulation Failures

3.1 Introduction

User access to a database is either an action to get some information from the database, or an action to give some information to the database in order to make it reflected by the database state. Access control is to impose restrictions on those actions in order to meet the requirements concerned with security. In many theoretical researches on security analysis, those two types of access are represented by read and write operations for simplicity. The basis of this simplification is the fact that any information flow between the users and the database originates in those operations. In practice, however, it is often the case that security requirements cannot be expressed in terms of such simple operations. For example, there may be a situation where a user should be allowed to get just partial information on some data but should not know the exact value of it. There is also a situation where a user should be allowed to update some data in some specific procedure, but should not be able to write any value he wants. We can impose these kinds of restrictions by defining appropriate functions and by authorizing users to invoke those functions instead of authorizing them to directly execute read or write operations. Those functions read the data but return a processed data through some computation, or they write the data following the required procedure. Although primitive read or write operations are invoked inside those functions, the users can invoke them only indirectly through those functions. In other words, the functions encapsulate those primitives into some procedures. By this encapsulation, we can achive context-dependent access control. The access control by utilizing encapsulation of functions (or "methods" in the object-oriented terminology) is one of important features of many object-oriented database systems [ADG92, Ber92, GGF93].

We use functions in order to encapsulate some sensitive primitive operations inside them. Therefore, it is essential that those functions certainly hide those primitives. If one can infer the result of a read operation encapsulated in some function, or can control (i.e. can change to any value he wants) the argument of a write operation inside some function, that function is not actually hiding those operations. The user

effectively obtains the same capability that he would have when he were allowed to invoke these primitives directly. Such flaws easily occur especially when the user exercises multiple capabilities together. For example, suppose a stock company has a database about all stockbrokers of the company. In this company, each stockbroker is given a budget for his stock dealing and there is a regulation that the budget of each broker should not be higher than ten times his salary. One clerk is assigned a job to periodically examine whether the budget of each broker is not illegally high against this regulation, but he should not be able to know the exact amount of the salary of each broker. Then, the database administrator defines a function that reads out the salary and the budget of a broker, compares them, and returns true or false. The clerk is authorized to invoke this function but is not authorized to directly read the salary data. In this situation, however, if that clerk can know the amount of the budget of some broker, he can know a little about the salary of the broker; "his salary is at least higher (lower) than this". Or in a worse case, if that user can change the amount of the budget to any value he wants, he can infer the exact amount of the salary by repeatedly changing the budget to several values and invoking the testing function. Those are security flaws. Another example is concerned with write access. Suppose the salary of each broker is updated once a week to a new value calculated from the budget given to him last week and the profit he made last week. Then, the database administrator defines a function that reads the budget and the profit of each broker, calculates a new salary value, and writes it in. An clerk is authorized to invoke this function. In this situation, if the employee is also able to change the budget of each broker to any value he wants, and as a consequence of it, can change the new salary value to any value he wants, then he can write any value he wants as the new salary. When we use encapsulation of functions for access control, many of these types of security flaws may occur.

In this chapter, we develop a technique to statically detect those flaws. We authorize users to invoke functions, and we also describe security requirements as negative authorizations like "one should not be able to infer the result of this read operation", or "one should not be able to control the argument of this write operation". We call the former capability *inferability* and call the latter capability *controllability*. These two capabilities effectively correspond to the abilities to invoke read or write operation directly. Because read and write operations can be considered as special cases of function invocations, we can naturally generalize the notion of inferability onto returned values of any functions and the notion of controllability onto arguments of any functions. In fact, we sometimes want to encapsulate a composed function into another function and to describe the requirement in terms of the encapsulated composed function, like "one should not be able to infer the result of this function invocation". Then, security requirements are described in the following forms; (1) the user u should not have inferability on the returned value of the function f, or (2) the user u should not have controllability on the argument a of the function f. Inferability on returned values of functions and controllability on arguments of functions precisely represent two kinds of user abilities in database access; (1) the ability to get data from the database and (2) the ability to give data to the database.

Based on these two notions, first, we establish a framework for description of security requirements. To define inferability formally, we design an inference system that models the inference ability of users. Next, we develop an algorithm that statically detects security flaws violating the given requirements. The main part of the algorithm is another inference system that simulates the inference system above by syntactical analysis of the program code of the functions.

As mentioned in [ADG92, Ber92, GGF93], when we control access in function granularity, there are two ways to impose restrictions; verifying only direct function invocations or verifying indirect invocations as well. The advantage of the former is that we can grant users encapsulated abstract operations, and the advantage of the latter is that security flaws are less likely to occur because every access is verified. We achieve both advantages by choosing the former approach and by providing the static security flaw detection mechanism.

3.1.1 Related Work

There are many researches on statically determining whether a user can infer sensitive information in the database especially in the context of relational database systems [Bur90, MJ88, JS91, SÖ91, HD92, QSK+93, Qia94]. Those researches focus on whether users can know the existence of some entities or can make sensitive associations between entities or values in the database, while we focus on different aspect, i.e. whether a user can compute the attribute values from supplied values. [Mor87, Row89, MSS88] propose frameworks to detect the possibility of user inference on sensitive values through the knowledge about semantic dependency or the integrity constraints between data in the database. On the other hand, our mechanism deals with dependencies between arguments or returned values of functions, and data in the database which are referred to in those functions. Although all researches deal with dependency, functions can represent wider range of dependencies than semantic dependencies or integrity constraints. In fact, our mechanisms can include integrity constraints by describing each integrity constraint in the form of a function with no argument and returning boolean values, and by assuming the result of the function is known to the user. On the contrary, a function cannot always be described in a form of constraint because the notion of functions include the notion of arguments, to which users can assign several values, and the notion of returned values of any types. Moreover, [Mor87, Row89] do not consider the effect of update by the user. [MSS88] takes into consideration only limited kind of situations; situations where users can infer some data by repeatedly updating some data. But their model does not analyze whether user can actually realize needed update, and does not include another kind of cases where update affects on inferability. In order to include the notion of arguments given by the user, and to examine more elaborately how the user can update the data, we introduce the notion of controllability and investigate how it interacts with inferability. We then develop an algorithm that analyzes both controllability and inferability, and show how that algorithm can be applied to static security flaw detection in object-oriented databases.

There are many researches on access control using "views" in object-oriented databases [TYI88, HZ90,

HO90, AB91, SLT91, Run92]. Also we propose mechanisms for access control with views in the previous chapter. Although the idea of access control using views defined by functions is essentially the same concept with access control using functions, those researches do not discuss about security issues. Our technique can be applied to the verification of view definitions in those systems as well. In [DAH+87], they provide a mechanism to automatically compute security levels of computed attributes in views in the context of the relational databases. Their method is, however, simply to compute the least upper bound of security levels of all data used in the computation without any analysis of the program code. Therefore, their method cannot be used for our purpose; to give users not total but partial information on some data through some computations.

[Den76, Coh77, DD77] proposed techniques to analyze program code in order to detect all flow of information. Their methods also compute the least upper bound of the security levels of source data. Therefore, their methods cannot be used for our purpose for the same reason mentioned above.

The rest of this chapter is organized as follows. Section 3.2 briefly explains the basic data model used as the base of the development. Section 3.3 discusses and formally defines the concept of inferability and controllability. Section 3.4 describes the algorithm that analyzes program code and detects security flaws. Finally, Section 3.5 mentions some further issues and concludes this chapter.

3.2 The Basic Model

In this section, we explain the basic model of database on which we develop our framework for access control. Although in this thesis we assume a simple data model with mutable objects and classes, the only essential point of the development shown in this thesis is that users access data objects by invoking functions. We believe that our mechanism can be translated onto several other data models, such as the relational data model with abstract data types.

The data model is defined as follows;

$$
\begin{aligned}
\mathit{schema} &= \langle \{\mathit{c\_name} : \mathit{o\_type}\},\ \{\langle \mathit{f\_name}(\mathit{arg} : \mathit{type}, \ldots, \mathit{arg} : \mathit{type}) : \mathit{type},\ \mathit{body}\rangle\},\ \{\mathit{v\_name} : \mathit{o\_type}\}\rangle \\
\mathit{o\_type} &= [\mathit{attribute} : \mathit{type}, \ldots, \mathit{attribute} : \mathit{type}] \\
\mathit{type} &= b \mid \mathit{c\_name} \mid \{\mathit{type}\} \\
\mathit{db} &= \langle \{\langle \mathit{c\_name}, \{\mathit{object}\}\rangle\},\ \{\langle \mathit{v\_name}, \mathit{object}\rangle\},\ \{\langle \mathit{u\_name},\ \{\mathit{f\_name} \mid \mathit{v\_name}\}\rangle\}\rangle
\end{aligned}
$$

A schema *schema* is a triple of a set of class definitions, a set of function definitions, a set of global variable definitions. Each class definition has the form $c\_name : o\_type$, which declares that the type of the instances of the class c_name is the type o_type. Each function definition is a pair of the signature of the form $f\_name(arg : type, \ldots, arg : type) : type$ and the definition of its body. Each global variable definition has the from $v\_name : o\_type$, which declares that the type of the object stored in the global variable v_name is the type o_type. The users access the database by invoking those functions. We call them *access functions*. We can interpret an access function also as a "method" defined on the class of the

first argument by interpreting the first argument as the receiver. Some additional consideration that would be needed if we introduced subtyping and overloading will be explained later. *o_type* $[a_1 : t_1, \ldots, a_n : t_n]$ denotes the type of objects with attributes $a_1, \ldots, a_n$ of type $t_1, \ldots, t_n$. Objects are mutable entities, and we can read out current values of their attributes, update their attributes, pass them to functions as arguments, and store them in attributes of other objects. *type* is either of a basic type b such as integer, a class name *c_name* that is interpreted as the type of its instances, or a set type of some type. A database *db* is a pair of (1) a set of pairs of a class name and its extension, i.e. a set of all objects that are instances of the class, and (2) a set of all registered user names, each of which is paired with his capability list. A capability list is a set of all access function names (or special functions explained later) that the user is allowed to invoke in the query, and all global variable names that the user is allowed to refer in the query. (The query language is defined later in this section.) Bodies of access functions are described using the *function definition language* defined by the following syntax;

$$e \quad ::= \quad c \mid a \mid x \mid X \mid f_b(e, \ldots, e) \mid f_a(e, \ldots, e) \mid \_r\_att(e) \mid \_w\_att(e, e) \mid \mathsf{let}\ x = e, \ldots, x = e\ \mathsf{in}\ e\ \mathsf{end}$$

c stands for constants, a stands for the arguments of the access function, x stands for local variables defined in some let construct), and X stands for global variables. $f_b(e, \ldots, e)$ is an invocation of a basic function f_b with arguments $e, \ldots, e$. Basic functions are primitive operations on basic types, such as addition on integers. $f_a(e, \ldots, e)$ is an invocation of another already defined access function f_a. *_r_att* and *_w_att* are special functions that read or write attributes of objects. For example, _r_salary(x) returns the current value of the attribute salary of the object x, and _w_salary(x, 100) writes 100 into the attribute salary of the object x and returns a special value null. let $x_1 = e_1, \ldots, x_n = e_n$ in e end is a construct that binds the result of the evaluation of $e_1, \ldots, e_n$ to $x_1, \ldots, x_n$, evaluates e under those binding, and return the result of that evaluation.

The users issue query using the following SQL-like query language;

select *item*, ..., *item* from $A_1 \in C_1, \ldots, A_n \in C_n$ where *condition*

In this syntax, $C_1, \ldots, C_n$ is a class name. $A_1, \ldots, A_n$ are called from-clause variables and are bound to each combination of instances of $C_1, \ldots, C_n$. *item* is $f(v, \ldots, v)$ where f is an access function or a special function (*_r_att* or *_w_att*) and v is either a constant of some basic type, some global variable name, or one of the from-clause variables. If object identifiers can have some printable form, such as ⟨id:730710⟩, we can also use them in place of v. In this development, however, we assume object identifiers do not have any printable form. We explain the reason of this choice later in Section 3.3. Items in a select clause are evaluated in order from left to right. *condition* consist of boolean terms connected by and and or where each boolean term has the form either of "$f(v, \ldots, v)$ *op* v" or "$f(v, \ldots, v)$ *op* $f(v, \ldots, v)$". *op* is a binary predicate for basic types, such as > for integers. For example, if the access function profile(x:*Person*):*string* is defined, and a user has _r_name, profile, and _r_age in his capability list, he can issue a query below;

select _r_name(p), profile(p) from p ∈ Person where _r_age(p) > 20

which returns a set of pairs of a name and a profile for all Person instances whose age are greater then 20. *item* can also be another nested select ... construct whose from clause includes, instead of class names, some set valued functions (or read operations of some set valued attributes). For example, suppose *Person* has an attribute child:{*Person*}. Then, the query below returns a set of names of children of a person with the name 'John';

```
select (select _r_name(q) from q ∈ child(p)) from p ∈ Person where _r_name(p) = 'John'
```

3.3 Specification of Security Requirements

In this section, we explain how we describe security requirements using the notion of inferability and controllability. We discuss the properties of those two kinds of capability and give the formal semantics of them.

3.3.1 Basic Concepts

In Section 3.1, we introduced the notion of inferability on returned values and controllability on arguments as ability effectively equivalent to ability to invoke the functions directly. While the meaning of inferability must be quite clear, the notion of controllability may need more explanation. Controllability on an argument intuitively means that one can use any value he wants as that argument. To have controllability on an argument, therefore, he must be able to change the value of that argument to any values. We call this ability *alterability*. Alterability only, however, is not enough to imply controllability. Suppose a function f is indirectly invoked in another function g. If one can change the value used as an argument of an function indirectly, but cannot know which value it takes now, it is similar to walking in the dark. He can move, but he cannot know where he is now. For example, suppose there is an access function f(x, o) defined as g(+(x, _r_age(o)) and one user is allowed to invoke f directly in the query. Then, he can change the value of the argument of g, i.e. the value of the expression +(x, _r_age(o)), to any values by changing the value of x. However, if he cannot know the value of _r_age(o), he cannot know the current value of the argument of g. We consider, therefore, having controllability equals to having both inferability and alterability. From now on, we decompose controllability into these two capabilities.

We further classify inferability into the ability to infer the exact value, *total inferability*, and the ability to infer a set of values to which the value must belong, *partial inferability*. Partial inferability is, in other words, the ability to infer at least one value that an expression can NOT be. Similarly, we classify alterability into the ability to change the value to any value of its type, *total alterability*, and the ability to change the value to only within some limited subset, *partial alterability*. Total inferability implies partial inferability, and total alterability do partial alterability.

Another dimension of classification of capability is certainty of them. If a user is always able to have a capability, in other words if he can have it no matter what the current database state is, we say that he

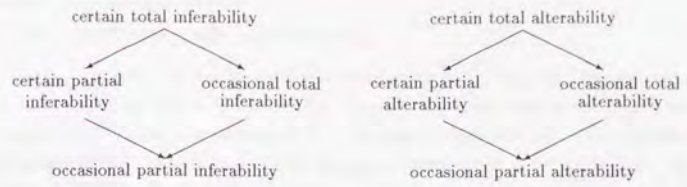


Figure 3.1: Implications between capabilities

has a certain capability. On the other hand, if a user is occasionally able to have a capability depending on the database state, we say that he has a occasional capability. Certain capability implies occasional capability.

Combining those two dimensions, we can consider four types of inferability; certain total inferability, certain partial inferability, occasional total inferability, and occasional partial inferability. Implication between them is illustrated in Figure 3.1. Similarly, alterability is classified into four types of alterability, and there are implications between them as illustrated in Figure 3.1.

Using these notions, we describe security requirements in the following syntax;

$$\begin{aligned} \textit{requirement} &::= (u, f(a_1 : \textit{caplist}, \ldots, a_n : \textit{caplist}) : \textit{caplist}) \\ \textit{caplist} &::= \textit{cap} : \textit{cap} : \ldots : \textit{cap} \\ \textit{cap} &::= \textbf{cti} \mid \textbf{cpi} \mid \textbf{cta} \mid \textbf{cpa} \mid \textbf{oti} \mid \textbf{opi} \mid \textbf{ota} \mid \textbf{opa} \end{aligned}$$

requirement $(u, f(a_1 : c_1^1 \cdots : c_1^{m_1}, \ \ldots \ , a_n : c_n^1 \cdots : c_n^{m_n}) : c_0^1 \cdots : c_0^{m_0}$ means that the user u should not be able to invoke the function f in a context where he can simultaneously achieve all specified capabilities c_i^j on each argument and on the returned value. f can be any of basic functions, access functions, or special functions explained in Section 3.2. Capabilities are certain total inferability **cti**, certain partial inferability **cpi**, certain total alterability **cta**, certain partial alterability **cpa**, or other four types of occasional capability. (It must be easy to imagine which capability each abbreviation represents.) We give the formal definition of the semantics of those capabilities and the description of security requirements later in this section. While the decomposition of controllability into inferability and alterability is essential for the security analysis described Section 3.4, it is not significant for security requirement description above. Indeed, we can include terms specifying controllability just as an abbreviation of specifying both inferability and alterability.

We show how security requirements are described using the examples explained in Section 3.1. Suppose the class Broker representing stockbrokers is defined with the type [name:*string*, salary:*int*, budget:*int*, profit:*int*]. The administrator defines the access function checkBudget as below;

```
checkBudget(broker:Broker):bool
    >=(_r_budget(broker), *(10, _r_salary(broker)))
```

The first line is the signature of this function, and the following line is the body. This function takes an argument broker, reads out budget and salary of it, compares the budget with ten times the salary, and returns true or false. A user u is authorized to invoke this function, but should not be able to know the exact amount of each broker's salary. This requirement is described as $(u, \_r\_salary(x) : \mathbf{cti})$. As we explained before, a flaw occurs if u can control (i.e. can infer and can alter) the value of _r_budget(broker). For example, if he also has _w_budget in his capability list, he can infer the salary of each broker by issuing the query below;

```
select _w_budget(b, 1), checkBudget(b), _w_budget(b, 2), checkBudget(b), ...
    from b ∈ Broker where _r_name(b) = 'John'
```

which will yield a set $\{\langle$null, false, ..., null, false, null, true, ...$\rangle\}$. (Here, we assume there is only one broker with the name 'John'. Of course, there may be more.) We detect this security flaw by examining whether the requirement above is satisfied or not. Similarly, the administrator defines the access function updateSalary as below;

```
updateSalary(broker:Broker):null
    _w_salary(broker, calcSalary(_r_budget(broker), _r_profit(broker)))
```

This function takes a broker object, read budget and profit of it, calculates his new salary by invoking another access function calcSalary, and writes the result to salary. A user u is authorized to invoke this access function but should not be able to change the new value of salary. In this case, however, if the user can alter the value of the expression _r_budget(broker), and as a consequence of it, can alter the returned value of calcSalary, he can alter the value of salary, which means there is a security flaw. This flaw can be detected by describing the security requirement $(u, \_w\_salary(a, v : \mathbf{cpa}))$, and by examining whether this requirement is satisfied or not.

If we introduced subtyping and overloading, a single access function name would correspond to multiple implementations defined on different subclasses. Even in such a situation, those multiple implementations are implementing a conceptually same functionality. Because it is natural to interpret those security requirements as referring not specific implementations but conceptual functionalities, we interpret each security requirement as specified for all the implementations of the function name. Under this interpretation, we can determine whether the requirement is satisfied or not by considering all possible subclasses. That process cannot be infinite because our language is recursion-free. If we also allowed recursion, the analysis would be quite hard, but that difficulty is common to various static analysis of a language with overloading and recursion.

Note that we assume the users can read the program code of the access functions. Even if we prohibit it, the users should know the semantics of the access functions to use them, and therefore, can infer the

contents of the access functions. To take that knowledge into consideration, we assume that the users can read the program code.

3.3.2 Causality between capabilities

As we can see in the examples above, in order to analyze inferability and alterability, we must consider inferability and alterability on every subexpression in the program code, and must trace how they are propagated to other subexpressions. Therefore, we generalize the notions of inferability and controllability to any expressions of the function definition language. Before giving formal semantics of them, we give some intuitive discussion about causality of those two kinds of capability.

There are two kinds of expressions, those of basic types and those of object types. First, we consider capability on basic type expressions.

The base cases of inferability on basic type expressions are expressions whose values are directly shown to the user, i.e. constants in the program code, or the arguments and the returned values of the functions directly invoked in the query. Inferability on basic type expressions is propagated through the "dependency" between expressions. This dependency comes from the dependency between arguments and returned values of basic functions. The type of inferability caused via dependency depends on the shape of the correspondence relation between the values of the expressions. Suppose the values of expressions e_1 and e_2 depends on each other, and in that dependency, each value of e_1 corresponds to max_1 values at maximum and to min_1 values at minimum. If $max_1 = 1$, certain total inferability on e_1 causes certain total inferability on the e_2, while if $1 < max_1 < |e_2|$ ($|e_2|$ denotes the number of values of domain of e_2), certain total inferability on e_1 causes only certain partial inferability on e_1. If $max_1 = |e_2|$, certain total inferability on e_1 causes no certain inferability on e_2. While certain inferability caused on e_2 is determined by max_1, occasional inferability is determined by min_1. Even when $max_1 > 1$, if $min_1 = 1$, occasional total inferability on e_1 causes occasional total inferability on e_2. Similarly, even when $max_1 = |e_2|$, if $min_1 < |e_2|$, then occasional total inferability on e_1 causes occasional partial inferability on e_2.

Another property that transmits inferability is equality. If the user knows that the values of two expressions must be the same, then inferability on either one causes inferability on the other. The user can know that the values of two expressions are equal if (1) they are the values stored in the same variable and no update occurs between those two expressions, or (2) they are the value of the same attribute of the same object and no update occurs between those two expressions.

Similarly, alterability on basic type expressions is propagated through dependency between returned values and arguments of basic functions. Suppose there is a expression $f(e_1, \ldots, e_n)$. If the returned value of f changes to all values of its type while the value of e_1 is changing to all values of its type, then total alterability on e_1 causes total alterability on $f(e_1, \ldots, e_n)$. If the returned value of f changes to only limited values, only partial alterability on $f(e_1, \ldots, e_n)$ is caused. If the returned value of f can change to all values only when the other arguments take specific values, then occasional total alterability

on $f(e_1, \ldots, e_n)$is caused. Alterability is also propagated through persistent data. If one has alterability on the argument of some write operation, he has alterability on the returned value of the following read operations that will read out the value written by that write operation. The user can alter the result of read operations also by changing the objects to be accessed, i.e. by utilizing alterability on the argument of the read operations. The base cases of alterability are expressions whose values are directly specified by the user, that is, the arguments of the functions directly invoked in the query.

Alterability can be analyzed independent of inferability because alterability is caused only by another alterability. Inferability never causes alterability. On the other hand, alterability plays a role to cause inferability. Suppose inferability on $e_1, \ldots, e_n$ causes certain partial inferability on e, that is, one can infer some set of values about e. If he can change $e_1, \ldots, e_n$ to several values, and can infer different subsets from different values of $e_1, \ldots, e_n$, then he can infer that the value of e must be in the intersection of those sets. If the intersection is a singleton set, he can infer the exact value of e. For example, suppose there is an expression >=(x, y). Certain total inferability on the whole expression and on x cause certain partial inferability on y. If one also has alterability on x, then he can infer the exact value of y by repeatedly changing x to several values and check the result of >=(x, y). This kind of situation is pointed out in [MSS88]. Another type of situations where alterability causes inferability is the case where inferability on some expression e causes inferability on another only when e takes some specific values. A typical example is the equality operator. Suppose there is an expression =(x, y) and one can infer the value of the whole expression and the value of x. They cause certain partial inferability on y. They also cause occasional total inferability because only when x happen to be equal to y, we can infer the exact value of y. If he has alterability on x, he can always make that situation happen, and therefore, he can always infer the value of y, that is, he has certain total inferability. Other examples of the former type of operations are the division or the reminder operator on integer. The division operator on integer is also an example of the latter type because only when the divisor is 1, total inferability on the result causes total inferability on the dividend, and vice versa.

Next, we consider capability on expressions of object types. The actual value of an object type expression is some form of object identifier. We can consider two kinds of situations. One is where object identifiers have some printable form, such as ⟨id:730710⟩, and another is where they do not have any printable form. In the former case, the user can directly specify an object in a query, and when the result of a query is an object, it can be output to output devices. In such a situation, capability on object identifiers can be treated in the same way as that on basic values. Because the development for this situation is rather simple, we assume the latter case in this development. In the latter case, when the result of a query was an object, it may be shown in some form like ⟨a Person object⟩, and the only way to pass an object to function in a query is to give it through from-clause variables or global variables. For example, in the example of query in Section 3.2, the argument for _r_name is given through the from-clause variable p. In such a situation, inferability on object identifiers does not make sense. Even in this situation, however, we can

recognize equality between two objects. For example, in the query in Section 3.2, we can recognize that p in _r_name(p) and p in profile(p) is identical within one evaluation of the select clause for one broker object. Similarly, in the access function checkBudget above, we can recognize that broker in _r_budget(broker) and broker in _r_salary(broker) are identical. As for alterability, we consider a user has total alterability on arguments of access functions directly invoked in a query even when the arguments are of object types. For example, in the query in Section 3.2, the user can alter an object passed to profile to any Person instance by changing the predicate in where clause (or rather he can invoke profile for all Person objects simply by using true in where clause). Therefore, he has alterability on the argument of profile.

3.3.3 Formalization

Now we give the formal semantics of the capability and the security requirement descriptions. We define the semantics of inferability by defining an inference system that formulates the inference ability of the users.

First, we clarify what a user can do in the queries with the select construct. In the select clause, a user can invoke the access functions or special functions in his capability list. Although he can get some information or can update some data within the where clause as well, what he can do in the where clause is a subset of what he can do in the select clause. Therefore, what a user can do in a query is essentially to invoke a sequence of functions in his capability list. In those invocations, as explained before, a user can pass arbitrary basic values and arbitrary objects as the arguments, and he can make two different arguments be the same object by using the same from-clause variable or the same global variable for those arguments.

To describe the semantics of capability, we introduce some notions. *A function sequence available to the user u* is a sequence of some functions in the capability list of u. These sequences correspond to what u can do in queries. $\langle f_1, \ldots, f_n \rangle$ denotes a function sequence consisting of $f_1, \ldots, f_n$. It may include the same function more than twice. $\mathcal{L}(u)$ denotes the set of all function sequences available to the user u. Given a sequence, we unfold each function in the sequence with replacing it to its body. We also recursively unfold all access function invocations inside it by replacing an access function invocation $f(e_1, \ldots, e_n)$ with let(f) $a_1 = e_1$, $a_n = e_n$ in e end, where e is a body of f and $a_1, \ldots, a_n$ are argument names of f. We use let(f) to record that this let construct was an invocation of f. Then, we number all subexpressions like ${}^k e$ where k is a serial number corresponding to the order of the evaluation in the actual execution of the sequence. $\mathcal{S}(L)$ denotes the set of all the numbered subexpressions in the sequence L. For example, if L = ⟨f, f⟩, f(x) = +(g(x),1), and g(y) = *(y,2), the unfolded and numbered sequence is:

$$\langle\ {}^{7}{+}({}^{5}\mathsf{let(g)\ y} = {}^{1}\mathsf{x\ in\ }{}^{4}{*}({}^{2}\mathsf{y},{}^{3}2)\ \mathsf{end},{}^{6}1),\ \ {}^{14}{+}({}^{12}\mathsf{let(g)\ y} = {}^{8}\mathsf{x\ in\ }{}^{11}{*}({}^{9}\mathsf{y},{}^{10}2)\ \mathsf{end},{}^{13}1)\rangle$$

and $\mathcal{S}(L)$ is

$$\{{}^{1}\mathsf{x},\ \ {}^{2}\mathsf{y},\ \ {}^{3}2,\ \ {}^{4}{*}({}^{2}\mathsf{y},{}^{3}2),$$

^{5}let(g) y = ^{1}x in ^{4}*(^{2}y,^{3}2) end, ^{6}1, ^{7}+(^{5}let(g) y = ^{1}x in ^{4}*(^{2}y,^{3}2) end, ^{6}1),
^{8}x, ^{9}y, ^{10}2, ^{11}*(^{9}y,^{10}2),
^{12}let(g) y = ^{8}x in ^{11}*(^{9}y,^{10}2) end, ^{13}1, ^{14}+(^{12}let(g) y = ^{8}x in ^{11}*(^{9}y,^{10}2) end,^{13}1) }

We number all subexpressions in order to distinguish different occurrences of the syntactically same subexpressions. If special functions are included in L, we number them without unfolding. For example, if _w_name(p, v) is included in L, we number it like ^{8}_w_name(^{6}p, ^{7}v). Because the signature of every access function is defined in the schema, the type of any subexpression can be statically determined. $Dom(^{k}e)$ denotes the set of all values of the type of ^{k}e.

Let $L = \langle f_1, \ldots, f_n \rangle$. Then *an execution instance of* L is a tuple of the form $\langle D,\ f_1(v_1^1, \ldots, v_1^{m_1}) = r_1,\ \ldots,\ f_n(v_n^1, \ldots, v_n^{m_n}) = r_n \rangle$, which represents an execution of L with D as the initial database state, with $v_1^1, \ldots, v_1^{m_1}, \ldots, v_n^1, \ldots, v_n^{m_n}$ as the arguments, and resulting to the returned values $r_1, \ldots, r_n$. If each r_i equals to the result of the actual execution, that execution instance is said to be *valid*. $\mathcal{E}(D, L)$ denotes the set of all valid execution instances of L with D as the initial database state. $[\![^{k}e]\!]_E$ denotes the value to which ^{k}e would be evaluated in the actual execution of the execution instance E. For example, let E be $\langle D,$ f(1)=3, f(2)=5$\rangle$, then E is a valid execution instance of L above, and $[\![^{3}*(^{1}\mathsf{x},^{2}2)]\!]_E = 2$.

Using these notations, we define the semantics of the security requirement descriptions as follows;

Definition 1 *A requirement* $(u, f(a_1 : c_1^1 \ldots : c_1^{m_1}, \ldots, a_n : c_n^1 \ldots : c_n^{m_n}) : c_0^1 \ldots : c_0^{m_0})$ *is not satisfied iff the following holds;*

$\exists L \in \mathcal{L}(u).\ \exists\ ^{k_0}let(f)\ a_1 =^{k_1} e_1, \ldots, a_n =^{k_n} e_n$ in ... end $\in \mathcal{S}(L)$.
$\{\forall i (1 \le i \le n).\ \forall j (1 \le j \le m_i).\ Can(L,\ c_i^j,\ ^{k_i}e_i)\}\ \wedge$
$\{\forall j (1 \le j \le m_0).\ Can(L,\ c_0^j,\ ^{k_0}let(f)$ ... end$)\}$ □

Here, $Can(L,\ c,\ ^{k}e)$, which is formally defined later, intuitively means that the capability c on ^{k}e can be achieved in L. This definition says if there exists a function sequence available to u which includes the subexpression $^{k_0}f(^{k_1}e_1, \ldots, ^{k_n}e_n)$, and if all the specified capabilities are achieved on that subexpression, then the requirement is not satisfied. If f is not an access function but a special function, then ^{k_0}let(f) $a_1 =^{k_1} e_1, \ldots, a_n =^{k_n} e_n$ in ... end in the definition above is replaced with $^{k_0}f(^{k_1}e_1, \ldots, ^{k_n}e_n)$. $Can(L,\ c,\ ^{k}e)$ is defined for each capabilities as follows;

Definition 2 $Can(L,$ **cta**, $^{k}e)$ *means the following holds;*

$\forall D.\ \forall v \in Dom(^{k}e).\ \exists E \in \mathcal{E}(D, L).\ [\![^{k}e]\!]_E = v$ □

Definition 3 $Can(L,$ **ota**, $^{k}e)$ *means the following holds;*

$\exists D.\ \forall v \in Dom(^{k}e).\ \exists E \in \mathcal{E}(D, L).\ [\![^{k}e]\!]_E = v$ □

Definition 4 $Can(L,$ **cpa**, $^{k}e)$ *means the following holds;*

$\forall D.\ \exists v_1, v_2 \in Dom(^{k}e).\ (v_1 \neq v_2)\ \forall v \in \{v_1, v_2\}.\ \exists E \in \mathcal{E}(D, L).\ [\![^{k}e]\!]_E = v$ □

Definition 5 *$Can(L,$* **opa**, *$^k e)$ means the following holds;*

$\exists D.\ \exists v_1, v_2 \in Dom(^k e).\ (v_1 \neq v_2)\ \forall v \in \{v_1, v_2\}.\ \exists E \in \mathcal{E}(D, L).\ [\![^k e]\!]_E = v$ □

Definition 6 *$Can(L,$* **cti**, *$^k e)$ means the following holds;*

$\forall D.\ \exists v \in Dom(^k e).\ \exists E \in \mathcal{E}(D, L).$

the inference system $\mathcal{I}(E)$ defined below can deduce $[^k e \in \{v\}]$ □

Definition 7 *$Can(L,$* **oti**, *$^k e)$ means the following holds;*

$\exists D.\ \exists v \in Dom(^k e).\ \exists E \in \mathcal{E}(D, L).$

the inference system $\mathcal{I}(E)$ defined below can deduce $[^k e \in \{v\}]$ □

Definition 8 *$Can(L,$* **cpi**, *$^k e)$ means the following holds;*

$\forall D.\ \exists S \subset Dom(^k e).\ \exists E \in \mathcal{E}(D, L).$

the inference system $\mathcal{I}(E)$ defined below can deduce $[^k e \in S]$ □

Definition 9 *$Can(L,$* **opi**, *$^k e)$ means the following holds;*

$\exists D.\ \exists S \subset Dom(^k e).\ \exists E \in \mathcal{E}(D, L).$

the inference system $\mathcal{I}(E)$ defined below can deduce $[^k e \in S]$ □

$S \subset S'$ means non-equal subset. The definition of total inferability says "no matter what the initial database state is, and for any value v in $Dom(^k e)$, one can make $^k e$ be evaluated to v by executing L with some arguments". The definition of partial inferability says "no matter what the initial database state is, there exist at least two values v_1 and v_2 in $Dom(^k e)$, and one can make $^k e$ be evaluated to v by executing L with some arguments". The definitions of occasional capabilities use $\exists D$ instead of $\forall D$, that is, they say "there exist a possible initial database state" instead of "no matter what the initial database state is". Inferabilities are defined with using $\mathcal{I}(E)$. $\mathcal{I}(E)$ is an inference system that formulates the inference that the users do from the observation of the execution of E. It performs inference on the terms defined by the syntax below:

$$term \ ::= \ [\langle {}^{k_1}e_1, \ldots, {}^{k_n}e_n\rangle \in S] \mid [{}^{k_1}e_1 = {}^{k_2}e_2]$$

$\langle {}^{k_1}e_1, \ldots, {}^{k_n}e_n\rangle$ denotes tuples of any number of expressions in $\mathcal{S}(E)$, and $[\langle {}^{k_1}e_1, \ldots, {}^{k_n}e_n\rangle \in S]$ is a term saying that $\langle [\![^{k_1}e_1]\!]_E, \ldots, [\![^{k_1}e_1]\!]_E\rangle \in S$. S is a subset of $Dom(^{k_1}e_1) \times \cdots \times Dom(^{k_n}e_n)$. $[^{k_1}e_1 = {}^{k_2}e_2]$ is a term saying that $[\![^{k_1}e_1]\!]_E$ equals to $[\![^{k_2}e_2]\!]_E$.

Let $E = \langle D, f_1(v_1^1, \ldots, v_1^{m_1}) = r_1,\ \ldots,\ f_n(v_n^1, \ldots, v_n^{m_n}) = r_n\rangle$. Then, the axioms and the inference rules of $\mathcal{I}(E)$ are as below. In that description, we use some macro expressions. When we use α and β (or α, β, and γ), it means that the inference rule holds for any A, B (or A, B, C) such that $A \cap B = \emptyset$ ($= B \cap C = C \cap A$) and $A \cup B$ ($\cup C$) $= \{1, \ldots, n\}$, and e_α means the row of e_α for all α in A. For example, if $A = \{1, 2\}$, then e_α means "e_1, e_2".

Definition 10 *The axioms and the inference rules of $\mathcal{I}(E)$*

(axioms)

$\rightarrow [\langle {}^{k}c\rangle \in \{c\}]$

$\rightarrow [\langle {}^{k}a_i^j\rangle \in \{v_i^j\}]$ (if a_i^j, the jth argument of f_i, has a basic type)

$\rightarrow [\langle {}^{k}e_i\rangle \in \{r_i\}]$ (if ${}^{k}e_i$, which is the entire body of f_i, has a basic type)

$\rightarrow [\langle {}^{k_1}\_r\_att({}^{k_2}e)\rangle \in Dom({}^{k_1}\_r\_att({}^{k_2}e))]$

$\rightarrow [\langle {}^{k_1}e_1, \ldots, {}^{k_n}e_n, {}^{k_0}f_b({}^{k_1}e_1, \ldots, {}^{k_n}e_n)\rangle \in \{\langle v_1, \ldots, v_n, r\rangle \mid f_b(v_1, \ldots, v_n) = r)\}]$

$\rightarrow [{}^{k_1}a = {}^{k_2}a]$ (${}^{k_1}a$ and ${}^{k_2}a$ are different occurrences of the same argument variable)

$\rightarrow [{}^{k_1}a_{i1}^{j1} = {}^{k_2}a_{i2}^{j2}]$ (if v_{i1}^{j1} and v_{i2}^{j2} are passed through the same from-clause variable)

$\rightarrow [{}^{k_1}a_{i1}^{j1} = {}^{k_2}X]$ (if v_{i1}^{j1} are passed through the global variable X)

$\rightarrow [{}^{k_1}X = {}^{k_2}X]$ (${}^{k_1}x$ and ${}^{k_2}x$ are different occurrences of the same global variable)

$\rightarrow [{}^{k_1}x = {}^{k_2}e]$ (if let $\ldots, x = {}^{k_2}e, \ldots$ in $\ldots$ end$\in \mathcal{S}(E)$ and ${}^{k_1}x$ is in the scope of that x)

$\rightarrow [{}^{k_1}e = {}^{k_2}$let $\ldots$ in ${}^{k_1}e$ end$]$

(inference rules)

1, join and projection of $\in$

$[\langle {}^{k_\alpha}e_\alpha, {}^{k_\beta}e_\beta\rangle \in S_1], [\langle {}^{k_\beta}e_\beta, {}^{k_\gamma}e_\gamma\rangle \in S_2] \rightarrow [\langle {}^{k_1}e_1, \ldots, {}^{k_n}e_n\rangle \in \{\langle v_1, \ldots, v_n\rangle \mid \langle v_\alpha, v_\beta\rangle \in S_1, \langle v_\beta, v_\gamma\rangle \in S_2\}]$

$[\langle {}^{k_1}e_1, \ldots, {}^{k_n}e_n\rangle \in S] \rightarrow [\langle {}^{k_\alpha}e_\alpha\rangle \in \{\langle v_\alpha\rangle \mid \exists \langle v_\beta\rangle.\ \langle v_1, \ldots, v_n\rangle \in S\}]$

2, rules for $=$

$[{}^{k_1}e_1 = {}^{k_2}e_2] \rightarrow [{}^{k_3}e_3 = {}^{k_4}\_r\_att({}^{k_2}e_2)]$ (if ^{k_5}_w_att$({}^{k_1}e_1, {}^{k_3}e_3) \in \mathcal{S}(E),\ k_5 < k_4$)

$[{}^{k_1}e_1 = {}^{k_2}e_2] \rightarrow [{}^{k_3}\_r\_att({}^{k_1}e_1) = {}^{k_4}\_r\_att({}^{k_2}e_2)]$

$[{}^{k_1}e_1 = {}^{k_2}e_2] \rightarrow [{}^{k_2}e_2 = {}^{k_1}e_1]$

$[{}^{k_1}e_1 = {}^{k_2}e_2], [{}^{k_2}e_2 = {}^{k_3}e_3] \rightarrow [{}^{k_1}e_1 = {}^{k_3}e_3]$

3, rule for $\in$ based on $=$

$[{}^{k_1}e_1 = {}^{k_2}e_2] \rightarrow [\langle {}^{k_1}e_1, {}^{k_2}e_2\rangle \in \{\langle v, v\rangle | v \in Dom({}^{k}_{1}e_1)\}]$

□

Base cases of terms with $\in$ are the constants in the program code c, the arguments v_i^j or the returned values r_i of the functions directly invoked by the user, and the constraints on the arguments and the returned values of the basic functions f_b. Starting those terms, the users proceed inference using the rules in 1. The users also use the knowledge on equality of two expressions. The rule 3 says that if two basic type expressions are equal, we can also use that information. The latter part of axioms say that two expressions are equal if they are the same argument variable, if they are different argument variables given values through the same from-clause variable, if they are a global variable and an argument variable given a value through that global variable, if they are a local variable and an expression bound to that variable in a let construct, or if they are a let construct and an expression between in and end of that let construct. The rules in 2 also say that two expressions are equal if they are values of the same attribute of the same object. If update has occurred, two values of the same attribute of the same object are not necessarily same. In

a model that allows multiple attributes to share an object, the users are not always able to determine whether an attribute has been updated or not, but sometimes they are. To include cases where they can determine it, we assume that the values of the same attribute of the same object are always same.

We show a simple example demonstrating how $\mathcal{I}(E)$ works. Consider the query below:

```
select _w_budget(b, 999), checkBudget(b), _w_budget(b, 1000), checkBudget(b)
    from b ∈ Broker where _r_name(b) = 'John'
```

and suppose that it yielded to $\{\langle$null, false, null, true$\rangle\}$. The function sequence corresponding this query is $\langle$_w_budget(o, v), checkBudget(a), _w_budget(o, v), checkBudget(a)$\rangle$, unfolded and numbered form of which is:

```
(3_w_budget(1o, 2v),
 10 >=(5_r_budget(4broker), 9*(610, 8_r_salary(7broker)))
 13_w_budget(11o, 12v),
 20 >=(15_r_budget(14broker), 19*(1610, 18_r_salary(17broker))))
```

and the execution instance corresponding this query E is:

```
⟨D, _w_budget(b, 999)=null, checkBudget(b)=false
_w_budget(b, 1000)=null, checkBudget(b)=true⟩
```

Then, $\mathcal{I}(E)$ can proceed the following inference:

$\rightarrow [\langle {}^{5}\text{\_r\_budget}(\ldots), {}^{9}*(\ldots), {}^{10}>=(\ldots)\rangle \in \{\langle 0,0,\text{true}\rangle, \langle 0,1,\text{false}\rangle, \ldots, \langle 999,999,\text{true}\rangle, \langle 999,1000,\text{false}\rangle, \ldots\}]$ (axiom for basic functions)

$\rightarrow [{}^{1}\text{o} = {}^{4}\text{broker}]$ (axiom for arguments)

$[{}^{1}\text{o} = {}^{4}\text{broker}] \rightarrow [{}^{2}\text{v} = {}^{5}\text{\_r\_budget}(\ldots)]$ (rule for $=$)

$[{}^{2}\text{v} = {}^{5}\text{\_r\_budget}(\ldots)] \rightarrow [\langle {}^{2}\text{v}, {}^{5}\text{\_r\_budget}(\ldots)\rangle \in \{\langle 0,0\rangle, \ldots, \langle 999,999\rangle, \ldots\}]$ (rule for $\in$ based on $=$)

$\rightarrow [{}^{2}\text{v} \in \{999\}]$ (axiom for arguments)

$[\langle {}^{2}\text{v}, {}^{5}\text{\_r\_budget}(\ldots)\rangle \in \{\langle 0,0\rangle, \ldots, \langle 999,999\rangle, \ldots\}], [{}^{2}\text{v} \in \{999\}] \rightarrow [\langle {}^{2}\text{v}, {}^{5}\text{\_r\_budget}(\ldots)\rangle \in \{\langle 999,999\rangle\}]$ (join)

$[\langle {}^{2}\text{v}, {}^{5}\text{\_r\_budget}(\ldots)\rangle \in \{\langle 999,999\rangle\}] \rightarrow [{}^{5}\text{\_r\_budget}(\ldots) \in \{999\}]$ (projection)

$[\langle{}^5\_r\_budget(\ldots),{}^9 *(\ldots),{}^{10} >= (\ldots)\rangle \in$
$\{\langle 0,0,\mathsf{true}\rangle, \langle 0,1,\mathsf{false}\rangle, \ldots, \langle 999,999,\mathsf{true}\rangle, \langle 999,1000,\mathsf{false}\rangle, \ldots\}],$
$[{}^5\_r\_budget(\ldots) \in \{999\}]$
$\quad \rightarrow [\langle{}^5\_r\_budget(\ldots),{}^9 *(\ldots),{}^{10} >= (\ldots)\rangle \in$
$\quad\quad \{\langle 999,0,\mathsf{true}\rangle, \ldots, \langle 999,999,\mathsf{true}\rangle, \langle 999,1000,\mathsf{false}\rangle, \ldots\}]$ (join)

$\rightarrow [{}^{10}>= (\ldots) \in \{\mathsf{false}\}]$ (axiom for returned values)

$[\langle{}^5\_r\_budget(\ldots),{}^9 *(\ldots),{}^{10} >= (\ldots)\rangle \in$
$\{\langle 999,0,\mathsf{true}\rangle, \ldots, \langle 999,999,\mathsf{true}\rangle, \langle 999,1000,\mathsf{false}\rangle, \ldots\}],$
$[{}^{10}>= (\ldots) \in \{\mathsf{false}\}]$
$\quad \rightarrow [\langle{}^5\_r\_budget(\ldots),{}^9 *(\ldots),{}^{10} >= (\ldots)\rangle \in$
$\quad\quad \{\langle 999,1000,\mathsf{false}\rangle, \ldots\}]$ (join)

$[\langle{}^5\_r\_budget(\ldots),{}^9 *(\ldots),{}^{10} >= (\ldots)\rangle \in$
$\{\langle 999,1000,\mathsf{false}\rangle, \ldots\}]$
$\quad \rightarrow [{}^9 *(\ldots) \in \{1000, \ldots\}]$ (projection)

$\rightarrow [{}^6 10 \in \{10\}]$ (axiom for constants)

$\rightarrow [\langle{}^6 10,{}^8\_r\_salary(\ldots),{}^9 *(\ldots)\rangle \in$
$\quad \{\langle 0,0,0\rangle, \langle 0,1,0\rangle, \ldots, \langle 10,0,0\rangle, \langle 10,1,10\rangle, \ldots\}]$ (axiom for basic functions)

$[{}^6 10 \in \{10\}],$
$[\langle{}^6 10,{}^8\_r\_salary(\ldots),{}^9 *(\ldots)\rangle \in$
$\{\langle 0,0,0\rangle, \langle 0,1,0\rangle, \ldots, \langle 10,0,0\rangle, \langle 10,1,10\rangle, \ldots\}]$
$\quad \rightarrow [\langle{}^6 10,{}^8\_r\_salary(\ldots),{}^9 *(\ldots)\rangle \in$
$\quad\quad \{\langle 10,0,0\rangle, \ldots, \langle 10,99,990\rangle, \langle 10,100,1000\rangle, \ldots\}]$ (join)

$[{}^9 *(\ldots) \in \{1000, \ldots\}],$
$[\langle{}^6 10,{}^8\_r\_salary(\ldots),{}^9 *(\ldots)\rangle \in$
$\{\langle 10,0,0\rangle, \ldots, \langle 10,99,990\rangle, \langle 10,100,1000\rangle, \ldots\}]$
$\quad \rightarrow [\langle{}^6 10,{}^8\_r\_salary(\ldots),{}^9 *(\ldots)\rangle \in$
$\quad\quad \{\langle 10,100,1000\rangle, \ldots\}]$ (join)

$[\langle{}^6 10,{}^8\_r\_salary(\ldots),{}^9 *(\ldots)\rangle \in \{\langle 10,100,1000\rangle, \ldots\}]$
$\quad \rightarrow [{}^8\_r\_salary(\ldots) \in \{100, \ldots\}]$ (projection)

$\rightarrow [\langle {}^{15}\_r\_budget(\ldots),{}^{19} *(\ldots),{}^{20} >= (\ldots)\rangle \in$
$\{\langle 0,0,\mathsf{true}\rangle, \langle 0,1,\mathsf{false}\rangle, \ldots, \langle 1000,1000,\mathsf{true}\rangle, \langle 1000,1001,\mathsf{false}\rangle, \ldots\}]$ (axiom for basic functions)

$\rightarrow [{}^{11}\mathsf{o} =^{14} \mathsf{broker}]$ (axiom for arguments)

$[{}^{11}\mathsf{o} =^{14} \mathsf{broker}] \rightarrow [{}^{12}\mathsf{v} =^{15} \_r\_budget(\ldots)]$ (rule for $=$)

$[{}^{12}\mathsf{v} =^{15} \_r\_budget(\ldots)]$
$\rightarrow [\langle {}^{12}\mathsf{v},{}^{15} \_r\_budget(\ldots)\rangle \in \{\langle 0,0\rangle, \ldots, \langle 1000,1000\rangle, \ldots\}]$ (rule for $\in$ based on $=$)

$\rightarrow [{}^{12}\mathsf{v} \in \{1000\}]$ (axiom for arguments)

$[\langle {}^{12}\mathsf{v},{}^{15} \_r\_budget(\ldots)\rangle \in \{\langle 0,0\rangle, \ldots, \langle 1000,1000\rangle, \ldots\}],\ [{}^{12}\mathsf{v} \in \{1000\}]$
$\rightarrow [\langle {}^{12}\mathsf{v},{}^{15} \_r\_budget(\ldots)\rangle \in \{\langle 1000,1000\rangle\}]$ (join)

$[\langle {}^{12}\mathsf{v},{}^{15} \_r\_budget(\ldots)\rangle \in \{\langle 1000,1000\rangle\}] \rightarrow [{}^{15} \_r\_budget(\ldots) \in \{1000\}]$ (projection)

$[\langle {}^{15}\_r\_budget(\ldots),{}^{19} *(\ldots),{}^{20} >= (\ldots)\rangle \in$
$\{\langle 0,0,\mathsf{true}\rangle, \langle 0,1,\mathsf{false}\rangle, \ldots, \langle 1000,1000,\mathsf{true}\rangle, \langle 1000,1001,\mathsf{false}\rangle, \ldots\}],$
$[{}^{15} \_r\_budget(\ldots) \in \{1000\}]$
$\rightarrow [\langle {}^{15} \_r\_budget(\ldots),{}^{19} *(\ldots),{}^{20} >= (\ldots)\rangle \in$
$\{\langle 1000,0,\mathsf{true}\rangle, \ldots, \langle 1000,1000,\mathsf{true}\rangle, \langle 1000,1001,\mathsf{false}\rangle, \ldots\}]$ (join)

$\rightarrow [{}^{20} >= (\ldots) \in \{\mathsf{true}\}]$ (axiom for returned values)

$[\langle {}^{15}\_r\_budget(\ldots),{}^{19} *(\ldots),{}^{20} >= (\ldots)\rangle \in$
$\{\langle 1000,0,\mathsf{true}\rangle, \ldots, \langle 1000,1000,\mathsf{true}\rangle, \langle 1000,1001,\mathsf{false}\rangle, \ldots\}],$
$[{}^{20} >= (\ldots) \in \{\mathsf{true}\}]$
$\rightarrow [\langle {}^{15} \_r\_budget(\ldots),{}^{19} *(\ldots),{}^{20} >= (\ldots)\rangle \in$
$\{\langle 1000,0,\mathsf{true}\rangle, \ldots, \langle 1000,1000,\mathsf{true}\rangle\}]$ (join)

$[\langle {}^{15}\_r\_budget(\ldots),{}^{19} *(\ldots),{}^{20} >= (\ldots)\rangle \in$
$\{\langle 1000,0,\mathsf{true}\rangle, \ldots, \langle 1000,1000,\mathsf{true}\rangle\}]$
$\rightarrow [{}^{19} *(\ldots) \in \{0,\ldots,1000\}]$ (projection)

$\rightarrow [{}^{16}10 \in \{10\}]$ (axiom for constants)

$\rightarrow [\langle {}^{16}10, {}^{18}\_\mathsf{r\_salary}(\ldots), {}^{19}*(\ldots)\rangle \in$
$\{\langle 0,0,0\rangle, \langle 0,1,0\rangle, \ldots, \langle 10,0,0\rangle, \langle 10,1,10\rangle, \ldots\}]$ (axiom for basic functions)

$[{}^{16}10 \in \{10\}],$
$[\langle {}^{16}10, {}^{18}\_\mathsf{r\_salary}(\ldots), {}^{19}*(\ldots)\rangle \in$
$\{\langle 0,0,0\rangle, \langle 0,1,0\rangle, \ldots, \langle 10,0,0\rangle, \langle 10,1,10\rangle, \ldots\}]$
$\rightarrow [\langle {}^{16}10, {}^{18}\_\mathsf{r\_salary}(\ldots), {}^{19}*(\ldots)\rangle \in$
$\{\langle 10,0,0\rangle, \ldots, \langle 10,100,1000\rangle, \langle 10,101,1010\rangle, \ldots\}]$ (join)

$[{}^{19}*(\ldots) \in \{0, \ldots, 1000\}],$
$[\langle {}^{16}10, {}^{18}\_\mathsf{r\_salary}(\ldots), {}^{19}*(\ldots)\rangle \in$
$\{\langle 10,0,0\rangle, \ldots, \langle 10,100,1000\rangle, \langle 10,101,1010\rangle, \ldots\}]$
$\rightarrow [\langle {}^{16}10, {}^{18}\_\mathsf{r\_salary}(\ldots), {}^{19}*(\ldots)\rangle \in$
$\{\langle 10,0,0\rangle, \ldots, \langle 10,100,1000\rangle\}]$ (join)

$[\langle {}^{16}10, {}^{18}\_\mathsf{r\_salary}(\ldots), {}^{19}*(\ldots)\rangle \in \{\langle 10,0,0\rangle, \ldots, \langle 10,100,1000\rangle\}]$
$\rightarrow [{}^{18}\_\mathsf{r\_salary}(\ldots) \in \{0, \ldots, 100\}]$ (projection)

$[{}^{18}\_\mathsf{r\_salary}(\ldots) =^{8} \_\mathsf{r\_salary}(\ldots)]$
$\rightarrow [\langle {}^{18}\_\mathsf{r\_salary}(\ldots), {}^{8}\_\mathsf{r\_salary}(\ldots)\rangle \in$
$\{\langle 0,0\rangle, \ldots, \langle 100,100\rangle, \ldots\}]$ (rule for $\in$ based on $=$)

$[\langle {}^{18}\_\mathsf{r\_salary}(\ldots), {}^{8}\_\mathsf{r\_salary}(\ldots)\rangle \in$
$\{\langle 0,0\rangle, \ldots, \langle 100,100\rangle, \ldots\}],$
$[{}^{18}\_\mathsf{r\_salary}(\ldots) \in \{0, \ldots, 100\}]$
$\rightarrow [\langle {}^{18}\_\mathsf{r\_salary}(\ldots), {}^{8}\_\mathsf{r\_salary}(\ldots)\rangle \in$
$\{\langle 0,0\rangle, \ldots, \langle 100,100\rangle\}]$ (join)

$[\langle {}^{18}\_\mathsf{r\_salary}(\ldots), {}^{8}\_\mathsf{r\_salary}(\ldots)\rangle \in$
$\{\langle 0,0\rangle, \ldots, \langle 100,100\rangle\}]$
$\rightarrow [{}^{8}\_\mathsf{r\_salary}(\ldots) \in \{0, \ldots, 100\}]$
(projection)

$[{}^{8}\_\mathsf{r\_salary}(\ldots) \in \{100, \ldots\}],\ [{}^{8}\_\mathsf{r\_salary}(\ldots) \in \{0, \ldots, 100\}]$
$\rightarrow [{}^{8}\_\mathsf{r\_salary}(\ldots) \in \{100\}]$ (join)

Thus, $\mathcal{I}(E)$ deduce $[{}^{8}\_\mathsf{r\_salary}(\ldots) \in \{100\}]$, whic corresponds to the fact that the user can infer the value

of the return value of ^{8}_r_salary(...).

3.4 An Algorithm for Flaw Detection

Although we gave formal semantics of security requirements in the last section, we cannot determine whether given requirements are satisfied or not by directly following that definition because that definition includes the notion of infinite sequences of functions. Examining all possible initial database state is also impractical. In this section, we develop an algorithm that detects security flaws in a reasonable amount of computation. This algorithm syntactically analyze program code using an inference system that simulates $\mathcal{I}(E)$ by taking some pessimistic assumptions. This algorithm is sound, that is, this algorithm always judges that the requirement is not satisfied when really it is not.

3.4.1 The Algorithm

In order to avoid infinite sequences of functions, our algorithm considers a set of all the functions included in the user's capability list, which must be a finite set, instead of considering a specific execution instance. We then develop an algorithm that determines whether there exists some function sequence consisting of only those functions in the set and further there exists its execution instance where the user can achieve each capabilities on each expressions. We then pessimistically assume that when the user can achieve some capabilities separately, there always exists some execution instance of some function sequence where he can achieve all those capabilities simultaneously. Based on this assumption, our algorithm just separately determines whether users can achieve each capability with using the given set of functions.

As a first step, in the same way we did for function sequences, we unfold all access function invocations in the given set of functions, and number all subexpressions like ${}_{i}e$. $\mathcal{S}'(F)$ denotes a set of all the numbered subexpressions in the set of function F. For example, let F be {f(x), _r_name(person)} and f(x) be _w_age(x, +(_r_age(x), 1)). Then unfolded and numbered functions are;

$$\{\ {}_{6}\text{\_w\_age}({}_{1}\text{x},\ {}_{5}\text{+}({}_{3}\text{\_r\_age}({}_{2}\text{x}),\ {}_{4}\text{1})),\ {}_{8}\text{\_r\_name}({}_{7}\text{person})\ \}$$

and $\mathcal{S}'(F)$ is

$$\{{}_{1}\text{x},\ {}_{2}\text{x},\ {}_{3}\text{\_r\_age}({}_{2}\text{x}),\ {}_{4}\text{1},\ {}_{5}\text{+}({}_{3}\text{\_r\_age}({}_{2}\text{x}),\ {}_{4}\text{1}),\ {}_{6}\text{\_w\_age}({}_{1}\text{x},\ {}_{5}\text{+}({}_{3}\text{\_r\_age}({}_{2}\text{x}),\ {}_{4}\text{1})),\ {}_{7}\text{person},\ {}_{8}\text{\_r\_name}({}_{7}\text{person})\ \}$$

Note that ${}_{i}e$ denotes each occurrence in some set F, and is different from ${}^{k}e$ denoting each occurrence in some sequence L. Each ${}^{k}e$ corresponds to one ${}_{i}e$ in $\mathcal{S}'(F)$ while each ${}_{i}e$ may correspond to multiple ${}^{k}e$ in one L. We call those ${}^{k}e$ *correspondent of* ${}_{i}e$. Then we develop an inference system $\mathcal{J}(F)$ that syntactically analyzes program code, and pessimistically determine whether the user can achieve each capability on each ${}_{i}e$. $\mathcal{J}(F)$ performs inference on terms defined by the following syntax;

$$\mathit{term} \ ::= \ \mathbf{cta}[{}_{i}e, num] \mid \mathbf{cpa}[{}_{i}e, num] \mid \mathbf{ota}[{}_{i}e, num] \mid \mathbf{opa}[{}_{i}e, num] \mid \mathbf{ceq}[{}_{i_1}e_1,\ {}_{i_2}e_2] \mid \mathbf{oeq}[{}_{i_1}e_1,\ {}_{i_2}e_2]$$

$$
\begin{aligned}
&\mid \mathbf{cti}[_le, \{\langle num, dir\rangle\}] \mid \mathbf{cpi}[\langle {}_{l_1}e_1, \ldots {}_{l_n}e_n\rangle, \{\langle num, dir\rangle\}] \\
&\mid \mathbf{oti}[_le, \{\langle num, dir\rangle\}] \mid \mathbf{opi}[\langle {}_{l_1}e_1, \ldots {}_{l_n}e_n\rangle, \{\langle num, dir\rangle\}] \\
&\mid =[{}_{l_1}e_1,\ {}_{l_2}e_2] \mid \mathbf{can}[_le,\ v] \\
dir \ ::= \ &+ \mid -
\end{aligned}
$$

cta, **cpa**, **ota**, and **opa** are terms saying that there may exist a function sequence L including ${}^k e$ which is correspondent of ${}_l e$ and on which the user achieves each capability. *num* in those terms is used to record where that alterability is achieved. The term for alterability is deduced in five ways; (1) the user can have alterability on argument variables of functions that he directly invokes, such as ${}_l a_k^j$. In this case, we record it with the number l. (2) The user may have alterability on the result of a basic function, such as ${}_l f(\ldots)$, through alterability on its arguments. In this case, we record it with the number l. (3) The user may have alterability on a read operation, such as ${}_{l_2}$r_att$({}_{l_1}e)$, through alterability on its argument, i.e. ${}_{l_1}e$. In this case, we record it with l_1. (4) The user may have alterability on a read operation also through alterability on arguments of some write operation to the same attribute, such as ${}_{l_2}e_2$ in ${}_{l_3}$w_att$({}_{l_1}e_1, {}_{l_2}e_2)$. In this case, we record it with the number l_2. (5) The user can alter the result of a read operation even through alterability on ${}_{l_1}e_1$ in some write operation ${}_{l_3}$w_att$({}_{l_1}e_1, {}_{l_2}e_2)$ because if he alter ${}_{l_1}e_1$ to the same object accessed in the read operation, the result of the read operation would be ${}_{l_2}e_2$, and if he alter ${}_{l_1}e_1$ to a different object, the result of the read operation could be a different value. In this case, we record it with the number l_1. *num* in terms on alterability works as follows; suppose there are expressions ^{5}w_age(^{3}a1, ^{4}i1), ^{12}w_age(^{10}a2, ^{11}i2), and ^{16}r_age(^{15}p). If the user can alter ^{3}a1 and ^{10}a2, he can alter the result of ^{16}r_age(^{15}p) by sometime letting ^{3}a1 be equal to ^{15}p and sometime letting ^{10}a2 be equal to ^{15}p. Further, if he has partial alterability on both ^{4}i and ^{11}i2, then the union of the ranges of each of them may happen to be equal to the entire domain of the type of the attribute, and thus he may have total alterability on ^{16}r_age(^{15}p). Similarly, if he has occasional alterability on both ^{4}i and ^{11}i2, then he may have certain alterability on ^{16}r_age(^{15}p). To find out such cases, we use *num* parameter of terms on alterability.

ceq$[{}_{l_1}e_1,\ {}_{l_2}e_2]$ is a term saying that there exists a function sequence L such that $\mathcal{S}(L)$ includes ${}^{k_1}e_1$ and ${}^{k_2}e_2$ which are correspondent of ${}_{l_1}e_1$ and ${}_{l_2}e_2$ respectively, and there exists its execution instance $E \in \mathcal{E}(()D, L)$ for any D where $[\![{}^{k_1}e_1]\!]_E = [\![{}^{k_2}e_2]\!]_E$. **oeq**$[{}_{l_1}e_1,\ {}_{l_2}e_2]$ is a similar term but it says there exists such a execution instance for some D. These terms are used to determine whether one can alter the result of a read operation by making the object accessed in that read operation and the object accessed in a proceeding write operation be equal. Therefore, we consider these terms only for expressions of object type.

cti and **oti** are terms saying that there may exist a function sequence L including ${}^k e$ which is correspondent of ${}_l e$ and on which the user achieves each capability. **cpi** and **opi** are similar, but they are defined on tuples of any number of expressions. They mean that $\mathcal{I}(E)$ may deduce $[\langle {}_{l_1}e_1, \ldots {}_{l_n}e_n\rangle \in S]$ with some S. $\{\langle num, dir\rangle\}$ is used to record where that inferability has been achieved. The reasons we

need to record it are explained later. Inferability originates in (1) inference on the returned value of some ${}_{l}f_{b}$ from the knowledge on its arguments, or (2) inference on the arguments of ${}_{l}f_{b}$ using the knowledge on its other arguments and its returned value. When the partial inferability is achieved through the former type of inference on ${}_{l}f_{b}$, we record it using l and $+$, and when it is achieved through the latter type of inference on ${}_{l}f_{b}$, we record it using k and $-$. We must use a set of pairs of *num* and *dir* because some capability may achieved joining multiple weaker capabilities as shown later.

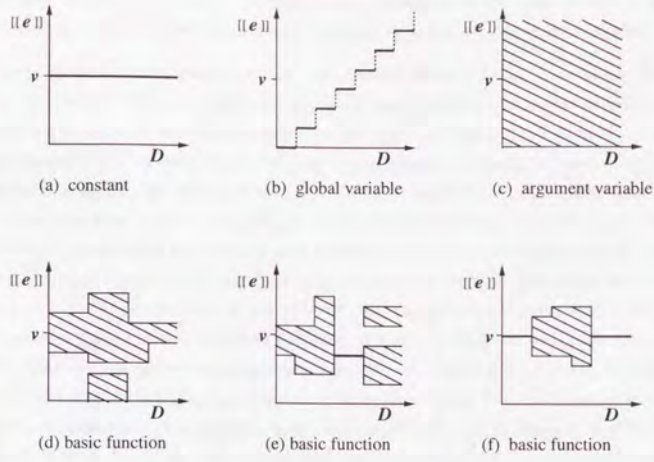
$=[{}_{l_1}e_1,\ {}_{l_2}e_2]$ is a term saying that there exists a function sequence and its execution instance where the users can deduce $[{}^{k_1}e_1\ =\ {}^{k_1}e_2]$. We need not distinguish whether it is always or just occasionally that one can deduce it because the deducability of $[{}^{k_1}e_1\ =\ {}^{k_1}e_2]$ is independent of the database state. Note that $=$ states that the user can know two expressions are equal, while **ceq** or **oeq** state that the user can let two expressions be evaluated to the same value, no matter whether he can know it or not.

$\mathbf{can}[{}_{l}e,\ v]$ is a term saying that there exists some ${}^{k}e$ which is a correspondent of ${}_{l}e$ and the user is always able to let ${}^{k}e$ be evaluated to v no matter what the current database state is. More formally, the term $\mathbf{can}[{}_{l}e,\ v]$ means the below may hold for some ${}^{k}e$ which is correspondent of ${}_{l}e$.

$$\exists L.\ \forall D.\ \exists E \in \mathcal{E}(D, L).\ {}^{k}e \in \mathcal{S}(L) \text{ and } [\![{}^{k}e]\!]_E = v$$

This term is used to test whether one can always achieve some capability or just occasionally. However, if we deduced this term for all the combination of ${}_{l}e$ and v with which the above holds, the number of the term that is to be deduced would be too large. For example, if the user have certain total alterability on ${}^{k}e$ which is correspondent of ${}_{l}e$, we must deduce $\mathbf{can}[{}_{l}e,\ v]$ for all values of $Dom({}^{k}e)$. Therefore, we deduce $\mathbf{can}[{}_{l}e,\ v]$ only when we don't have certain total/partial inferability on it but the above hold. There are two cases where this happens. One is a case where ${}_{l}e$ constantly takes v, and the other is a case where the user has occasional alterability on ${}_{l}e$, and when he cannot alter the value, ${}_{l}e$ takes c. An typical example of the latter case is *(X, a) where the user have no alterability on X but have total alterability on argument variable a. In this case, **can**[*(X, a), 0] holds because the user has occasional alterability on *(X, a) (only when $X = 0$, he cannot alter the value), and when $X = 0$, *(X, a) takes 0.

Figure 3.2 helps us understand those situations. These charts illustrates which range of values an expression e takes while the initial database state D changes. The horizontal axis of each chart is the initial database state D and the vertical axis is the value of ${}^{k}e$ in some $\mathcal{S}(L)$. We plot points (D, v) such that there exists an execution instance $E(D, L)$ where $[\![{}^{k}e]\!]_E = v$. Chart (a) illustrates the case where ${}^{k}e$ is a constant expression ${}^{k}c$. In this case, the user does not have any alterability. Chart (b) illustrates the case for global variable X. In this case the user also does not have any alterability, but the value changes while the initial database state changes. The chart for a read operation, such as ^{k_1}r_age(^{k_2}p), is also like (b) if there is no update in L before the read operation. If there is an update writing e' as its new value, then the chart for the read operation is the same with the chart for e'. Chart (c) illustrates the case for argument variables of functions directly invoked in L. In this case, the user can pass any value to the variable, that is, he have certain total alterability. The charts for basic functions can be various forms



(a) constant (b) global variable (c) argument variable

(d) basic function (e) basic function (f) basic function

Figure 3.2: Actual ranges of values of expressions

depending on the situations, such as (d), (e), or (f). The statement above holds in (a), (c), (d), and (f), but we deduce $\mathbf{can}[^k e,\ v]$ only for (a) and (f).

When defining the axioms and the inference rules of $\mathcal{J}(F)$, we take the following pessimistic assumptions;

- for any $^{k_1}e_1, \ldots, ^{k_n}e_n$, there is a possibility that $\{\langle v_1, \ldots, v_n\rangle \mid \exists D.\ \exists E \in \mathcal{E}(D, L).\ [\![^k e_1]\!]_E = v_1, \ldots, [\![^k e_n]\!]_E = v_n\}$ is a set with only two elements.

- if $\mathcal{I}(E)$ can deduce $[^k e \in S]$ and $[^k e \in S']$ through two different ways, $S \cap S'$ may be a singleton set. In other words, if a user can achieve partial inferability through two different ways, he may have total inferability. This is one case where one can achieve some capability by joining multiple weaker capabilities.

- if one can have an occasional inferability through two different ways, he may always be able to infer it through either way, that is, he may have a certain inferability. This is another case where one can achieve some capability by joining multiple weaker capabilities.

- if one can have an occasional partial alterability through two different ways, he may have certain total alterability. This also is a case where one can achieve some capability by joining multiple weaker capabilities.

- when there are two read operations (or one write operation and one read operation) on the same attributes, update obstructing the inference does not occur between those two operations.

We show some examples where those pessimistic assumptions actually holds. An example of the second assumptions (expressions that takes only two values) is a basic function that gets an integer i and return the i mod 2. An example of the situation where the user can have certain total alterability through two different occasional partial alterability is as follows. Suppose there is a read operation _r_age(X) and two write operations _w_age(a1, f1(Y,a2)) and _w_age(a3, f2(Y,a4)) where a1,...,a4 are argument variables of directly invoked functions, X and Y are global variables, and f1 and f2 are basic functions. Suppose the user can alter the result of f1(Y,a2) to any integer but 0 by changing the value of a2 when Y is not 0 while f1(Y,a2) takes 0 when Y is 0. On the other hand, he can alter the result of f2(Y,a4) to any integer but 0 by changing the value of a4 only when Y is 0 while f2(Y,a4) takes 0 when Y is not 0. In other words, the user has occasional partial alterability on f1(Y,a2) and f2(Y,a4). In addition, because he can make either a2 or a4 be equal to X, he has occasional partial alterability on _r_age(a1) through two different ways. In this situation, the user can indeed have certain total alterability. When Y is not 0, he can let the value of _r_age(X) be any value but 0 by assigning X to a1, and can let the value of _r_age(X) be 0 by assigning X to a2. When Y is 0, he can let the value be any value but 0 by assigning X to a2, and can let the value of _r_age(X) be 0 by assigning X to a1.

On the contrary, one can never achieve certain total inferability through two occasional partial inferability. Suppose one has occasional partial inferability through two ways. Then there exists a database state D_1 where one cannot have partial inferability in one way. Therefore, he cannot have total inferability when the database state is D_1 even if he can have partial inferability through the other way at that time.

The axioms and the inference rules of $\mathcal{J}(F)$ is defined as below.

Definition 11 *The axioms and the inference rules of* $\mathcal{J}(F)$

(axioms and rules for alterability)

$\rightarrow \mathbf{cta}[_{l}a, l]$ (a is an argument variable of an outer-most function)

$\mathbf{opa}[_{l_1}e_1, l_3] \rightarrow \mathbf{ota}[_{l_2}\_r\_att(_{l_1}e_1), l_1]$

$\mathbf{opa}[_{l_1}e_1],\ \mathbf{oeq}[_{l_1}e_1,\ _{l_2}e_2] \rightarrow \mathbf{ota}[_{l_4}\_r\_att(_{l_2}e_2), l_5]$ (if $_{l_5}\_w\_att(_{l_1}e_1,\ _{l_3}e_3) \in \mathcal{S}'(F)$)

$\mathbf{ceq}[_{l_1}e_1,\ _{l_2}e_2],\ \mathbf{cta}[_{l_3}e_3, l_4] \rightarrow \mathbf{cta}[_{l_5}\_r\_att(_{l_2}e_2), l_4]$ (if $_{l_6}\_w\_att(_{l_1}e_1,\ _{l_3}e_3) \in \mathcal{S}'(F)$)

$\mathbf{ceq}[_{l_1}e_1,\ _{l_2}e_2],\ \mathbf{cpa}[_{l_3}e_3, l_4] \rightarrow \mathbf{cpa}[_{l_5}\_r\_att(_{l_2}e_2), l_4]$ (if $_{l_6}\_w\_att(_{l_1}e_1,\ _{l_3}e_3) \in \mathcal{S}'(F)$)

$\mathbf{oeq}[_{l_1}e_1,\ _{l_2}e_2],\ \mathbf{ota}[_{l_3}e_3, l_4] \rightarrow \mathbf{ota}[_{l_5}\_r\_att(_{l_2}e_2), l_4]$ (if $_{l_6}\_w\_att(_{l_1}e_1,\ _{l_3}e_3) \in \mathcal{S}'(F)$)

$\mathbf{oeq}[_{l_1}e_1,\ _{l_2}e_2],\ \mathbf{opa}[_{l_3}e_3, l_4] \rightarrow \mathbf{opa}[_{l_5}\_r\_att(_{l_2}e_2), l_4]$ (if $_{l_6}\_w\_att(_{l_1}e_1,\ _{l_3}e_3) \in \mathcal{S}'(F)$)

$\mathbf{cta}[_{l_1}e, l_3] \rightarrow \mathbf{cta}[_{l_2}x, l_3]$

(if let $\ldots, x = {}_{l_1}e, \ldots$ in $\ldots$ end $\in \mathcal{S}(F)$ and ${}_{l_2}x$ in in the scope of that x)

$\mathbf{cpa}[_{l_1}e, l_3] \rightarrow \mathbf{cpa}[_{l_2}x, l_3]$

(if let $\ldots, x = {}_{l_1}e, \ldots$ in $\ldots$ end $\in \mathcal{S}(F)$ and ${}_{l_2}x$ in in the scope of that x)

$\mathbf{ota}[_{l_1}e, l_3] \rightarrow \mathbf{ota}[_{l_2}x, l_3]$

(if let $\ldots, x = {}_{l_1}e, \ldots$ in $\ldots$ end $\in \mathcal{S}(F)$ and ${}_{l_2}x$ in in the scope of that x)

$\mathbf{opa}[_{l_1}e, l_3] \rightarrow \mathbf{opa}[_{l_2}x, l_3]$

(if let $\ldots, x = {}_{l_1}e, \ldots$ in $\ldots$ end $\in \mathcal{S}(F)$ and ${}_{l_2}x$ in in the scope of that x)

$\mathbf{cta}[_{l_1}e, l_3] \rightarrow \mathbf{cta}[_{l_2}$let $\ldots$ in ${}_{l_1}e$ end$, l_3]$

$\mathbf{cpa}[_{l_1}e, l_3] \rightarrow \mathbf{cpa}[_{l_2}$let $\ldots$ in ${}_{l_1}e$ end$, l_3]$

$\mathbf{ota}[_{l_1}e, l_3] \rightarrow \mathbf{ota}[_{l_2}$let $\ldots$ in ${}_{l_1}e$ end$, l_3]$

$\mathbf{opa}[_{l_1}e, l_3] \rightarrow \mathbf{opa}[_{l_2}$let $\ldots$ in ${}_{l_1}e$ end$, l_3]$

$\mathbf{opa}[_{l}e, l_1], \mathbf{opa}[_{l}e, l_2] \rightarrow \mathbf{cta}[_{l}e, l_1] \qquad (l_1 \neq l_2)$

(axioms and rules for **ceq** and **oeq**)

$=[_{l_1}e_1,\ {}_{l_2}e_2] \rightarrow \mathbf{ceq}[_{l_1}e_1,\ {}_{l_2}e_2]$

$\rightarrow \mathbf{ceq}[_{l_1}a,\ {}_{l_2}e]$

(if a is an argument variable of outer-most function, and a and e have the same object type)

$\rightarrow \mathbf{oeq}[_{l_1}e_1,\ {}_{l_2}e_2]$ (if ${}_{l_1}e_1$ and ${}_{l_2}e_2$ have the same object type)

$\mathbf{ceq}[_{l_1}e_1,\ {}_{l_2}e_2] \rightarrow \mathbf{ceq}[_{l_4}\_r\_att(_{l_2}e_2),\ {}_{l_3}e_3]$ (if ${}_{l_8}\_w\_att(_{l_1}e_1,\ {}_{l_3}e_3) \in \mathcal{S}'(F)$)

$\mathbf{oeq}[_{l_1}e_1,\ {}_{l_2}e_2] \rightarrow \mathbf{oeq}[_{l_4}\_r\_att(_{l_2}e_2),\ {}_{l_3}e_3]$ (if ${}_{l_5}\_w\_att(_{l_1}e_1,\ {}_{l_3}e_3) \in \mathcal{S}'(F)$)

$\mathbf{ceq}[_{l_1}e_1,\ {}_{l_2}e_2] \rightarrow \mathbf{ceq}[_{l_3}\_r\_att(_{l_1}e_1),\ {}_{l_4}\_r\_att(_{l_2}e_2)]$

$\mathbf{oeq}[_{l_1}e_1,\ {}_{l_2}e_2] \rightarrow \mathbf{oeq}[_{l_3}\_r\_att(_{l_1}e_1),\ {}_{l_4}\_r\_att(_{l_2}e_2)]$

$\mathbf{ceq}[_{l_1}e_1,\ {}_{l_2}e_2] \rightarrow \mathbf{ceq}[_{l_2}e_2,\ {}_{l_1}e_1]$

$\mathbf{oeq}[_{l_1}e_1,\ {}_{l_2}e_2] \rightarrow \mathbf{oeq}[_{l_2}e_2,\ {}_{l_1}e_1]$

$\mathbf{ceq}[_{l_1}e_1,\ {}_{l_2}e_2],\ \mathbf{ceq}[_{l_2}e_2,\ {}_{l_3}e_3] \rightarrow \mathbf{ceq}[_{l_1}e_1,\ {}_{l_3}e_3]$

$\mathbf{oeq}[_{l_1}e_1,\ {}_{l_2}e_2],\ \mathbf{oeq}[_{l_2}e_2,\ {}_{l_3}e_3] \rightarrow \mathbf{oeq}[_{l_1}e_1,\ {}_{l_3}e_3]$

(axioms and rules for inferability)

$\rightarrow \mathbf{cti}[_{l}c,\ \{\langle l,\ +\rangle\}]$

$\rightarrow \mathbf{cti}[_{l}a,\ \{\langle l,\ +\rangle\}]$ (a is an argument variable of an outer-most function)

$\rightarrow \mathbf{cti}[_{l}e,\ \{\langle l,\ -\rangle\}]$ (${}_{l}e$ is the entire body of an outer-most function)

$=[e_1,\ e_2] \rightarrow \mathbf{cpi}[\langle e_1, e_2\rangle,\ \{\langle 0,\ +\rangle\}]$

$=[e_1,\ e_2],\ \mathbf{cti}[e_1,\ S] \rightarrow \mathbf{cti}[e_2,\ S]$

$=[e_1,\ e_2],\ \mathbf{cpi}[\langle \ldots, e_1, \ldots\rangle,\ S] \rightarrow \mathbf{cpi}[\langle \ldots, e_2, \ldots\rangle,\ S]$

$=[e_1,\ e_2],\ \mathbf{oti}[e_1,\ S] \rightarrow \mathbf{oti}[e_2,\ S]$

$=[e_1,\ e_2],\ \mathbf{opi}[\langle \ldots, e_1, \ldots\rangle,\ S] \rightarrow \mathbf{opi}[\langle \ldots, e_2, \ldots\rangle,\ S]$

$\mathbf{cpi}[\langle e_\alpha, e_\beta\rangle,\ S_1],\ \mathbf{cpi}[\langle e_\beta, e_i, \ldots\rangle,\ S_2] \rightarrow \mathbf{cpi}[\langle e_\alpha, e_i\rangle,\ S] \quad (S_1 \cap S_2 = \emptyset)$

$\mathbf{cpi}[_l e,\ S_1],\ \mathbf{cpi}[_l e,\ S_2] \rightarrow \mathbf{cti}[_l e,\ S_1 \cup S_2] \quad (S_1 \setminus S_2 \neq \emptyset \ \wedge\ S_2 \setminus S_1 \neq \emptyset)$

$\mathbf{oti}[_l e,\ S_1],\ \mathbf{oti}[_l e,\ S_2] \rightarrow \mathbf{cti}[_l e,\ S_1 \cup S_2] \quad (S_1 \setminus S_2 \neq \emptyset \ \wedge\ S_2 \setminus S_1 \neq \emptyset)$

$\mathbf{opi}[_l e,\ S_1],\ \mathbf{opi}[_l e,\ S_2] \rightarrow \mathbf{cpi}[_l e,\ S_1 \cup S_2] \quad (S_1 \setminus S_2 \neq \emptyset \ \wedge\ S_2 \setminus S_1 \neq \emptyset)$

$\mathbf{opi}[_l e,\ S_1],\ \mathbf{opi}[_l e,\ S_2] \rightarrow \mathbf{oti}[_l e,\ S_1 \cup S_2] \quad (S_1 \setminus S_2 \neq \emptyset \ \wedge\ S_2 \setminus S_1 \neq \emptyset)$

(axioms and rules for $=$)

$\rightarrow =[_{l_1} a,\ _{l_2} a]$ ($_{l_1}a$ and $_{l_2}a$ are different occurrences of the same argument variable)

$\rightarrow =[_{l_1} a_1,\ _{l_2} a_2]$ (a_1 and a_2 are argument variables of outer-most functions of the same type)

$\rightarrow =[_{l_1} a,\ _{l_2} X]$ (a is an argument variable of an outer-most function of the same type)

$\rightarrow =[_{l_1} X,\ _{l_2} X]$ ($_{l_1}X$ and $_{l_2}X$ are different occurrences of the same global variable)

$\rightarrow =[_{l_1} x,\ _{l_2} e]$ (if $\mathsf{let}\ \ldots, x = {}_{l_2}e, \ldots\ \mathsf{in}\ \ldots\ \mathsf{end} \in \mathcal{S}(F)$ and $_{l_1}x$ in the scope of that x)

$\rightarrow =[_{l_1} e,\ _{l_2}\mathsf{let}\ \ldots\ \mathsf{in}\ {}_{l_1}e\ \mathsf{end}]$

$=[e_1,\ e_2] \rightarrow =[e_3,\ \_r\_att(e_2)]$ (if $\_w\_att(e_1, e_3) \in \mathcal{S}'(F)$)

$=[_{l_1} e_1,\ _{l_2} e_2] \rightarrow =[_{l_3}\_r\_att(_{l_1} e_1),\ _{l_4}\_r\_att(_{l_2} e_2)]$

$=[e_1,\ e_2] \rightarrow =[e_2,\ e_1]$

$=[e_1,\ e_2],\ =[e_2,\ e_3] \rightarrow =[e_1,\ e_3]$

(axiom for **can**)

$\rightarrow \mathbf{can}[_l e,\ c]$

(rules for implications between capabilities)

$\mathbf{cti}[_l e,\ S] \rightarrow \mathbf{cpi}[_l e,\ S]$

$\mathbf{cti}[_l e,\ S] \rightarrow \mathbf{oti}[_l e,\ S]$

$\mathbf{oti}[_l e,\ S] \rightarrow \mathbf{opi}[_l e,\ S]$

$\mathbf{cpi}[_l e,\ S] \rightarrow \mathbf{opi}[_l e,\ S]$

$\mathbf{cta}[_l e,\ S] \rightarrow \mathbf{cpa}[_l e,\ S]$

$\mathbf{cta}[_l e,\ S] \rightarrow \mathbf{ota}[_l e,\ S]$

$\mathbf{ota}[_l e,\ S] \rightarrow \mathbf{opa}[_l e,\ S]$

$\mathbf{cpa}[_l e,\ S] \rightarrow \mathbf{opa}[_l e,\ S]$

□

Most rules for equality and inferability directly corresponds to rules of $\mathcal{I}(L)$. The rule $\mathbf{cpi}[e, S_1], \mathbf{cpi}[e, S_2] \rightarrow \mathbf{cti}[e, S_1 \cup S_2]$ corresponds to the rule of join of $[\langle e \rangle \in S]$, and represents that we take a pessimistic assumption that the intersection of two different subsets can be always a singleton set.

We need to record how a inferability is caused for two reasons. The first reason is that we assumed that if the user can deduce $[^k e \in S]$ with two "different" ways, the intersection of those two S can be a singleton set. Therefore, we need to record how each partial inferability is caused. The second reason is that we should not feed back a inferability to its cause. For example, suppose there is an expression +(X, 1), and the user has partial inferability on X. Then, the user can have partial inferability on +(X, 1). When there is a expression +(a, 1) and a user has partial inferability on +(a, 1), he can achieve inferability on a. Therefore, in the above case, the user can achieve partial inferability on X again via inferability on +(X, 1). This means the user can achieve partial inferability on X in two different ways. However, those two sets of values inferred for X are exactly the same, and so he never can have total inferability on X. What was wrong is to feed back the inferability caused by X to itself again.

In addition to those rules, we must also specify the rules based on basic functions. Because those rules depend on the semantics of each basic function, we must define those rules by hand. We can, however, provide "metarules" of the form "if the semantics of the basic function f_b satisfies this condition, then this rule must be added". Only if rules on basic functions are correctly given following those metarules, our algorithm can analyze any composed functions. We list the metarules below. The rules listed below are general form that can be applied to basic functions with any number of arguments, denoted as $f_b(a_1, \ldots a_n)$ in the list below. In the description below, the lines beginning with "if" is conditions on the semantics, and the following lines specify rules to be added. To denote metarules, we use some new notations. $\bowtie$ and Π denotes join operation and projection operation used in $\mathcal{I}$. A new word "complete subset" is also used. We say a set S is a complete subset of $S_1 \times \ldots \times S_n$ iff $S \subset S_1 \times \ldots \times S_n$ and $\Pi(i)S = S_i$ for any i. In order to describe the metarules in general form that can be applied to functions with any number of arguments, we use some macro expressions. When we use $\mathcal{A}$, $\mathcal{B}$, ... in a description, it means that the metarule holds for any A, B, ... such that $A \cap B = A \cap C = \ldots = \emptyset$ and $A \cup B \cup \ldots = \{1, \ldots, n\}$, and if we write $\mathcal{A}(\textit{statement})(\textit{delimiter})$, it means the row of $\textit{statement}[i/\mathrm{A}]$ for all i in A with a delimiter between each elements. When no delimiter is needed, we omit the second pair of parentheses. For example, if $A = \{3, 4, 7\}$ and we write "$\mathcal{A}(\exists v'_{\mathcal{A}}(\neq v_{\mathcal{A}}).)$", it is an abbreviation of "$\exists v'_3(\neq v_3).\ \exists v'_4(\neq v_4).\ \exists v'_7(\neq v_7).$" and "$\mathcal{A}(\mathit{Dom}(a_{\mathcal{A}}))(\times)$" is an abbreviation of "$\mathit{Dom}(a_3) \times \mathit{Dom}(a_4) \times \mathit{Dom}(a_7)$". $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{R}$, ... are also used in the same way and they represent any subset of $\{1, \ldots, n\}$. We also abbreviate "$\exists vv_1, vv_2\ (vv_1 \neq vv_2)\ \in S.\ \forall v \in \{vv_1, vv_2\}$" to "$\exists 2(v \in S)$".

1, metarules for alterability

if $\mathcal{C}(\exists v_{\mathcal{C}}.)\ \mathcal{D}(\exists vv_{\mathcal{D}}, vv'_{\mathcal{D}}\ (vv_{\mathcal{D}} \neq vv'_{\mathcal{D}})\ \in \mathit{Dom}(a_{\mathcal{D}}).)\ \forall \langle \mathcal{D}(v_{\mathcal{D}})(,) \rangle \in \{\langle \mathcal{D}(vv_{\mathcal{D}})(,) \rangle, \langle \mathcal{D}(vv_{\mathcal{D}})(,) \rangle\}.$
$\mathcal{B}(\exists v'_{\mathcal{B}}.)\ \forall r \in \mathit{Dom}(f_b).\ \mathcal{A}(\exists v_{\mathcal{A}}.)\ \mathcal{B}(\exists v_{\mathcal{B}}\ (\neq v'_{\mathcal{B}}).)$
$f_b(v_1, \ldots, v_n) = r$

$\mathcal{A}(\mathbf{cta}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,),\ \mathcal{B}(\mathbf{cpa}[e_{\mathcal{B}}, l_{\mathcal{B}}])(,),\ \mathcal{C}(\mathbf{can}[e_{\mathcal{C}},\ v_{\mathcal{C}}])(,) \rightarrow \mathbf{cta}[_{l} f_b(e_1, \ldots, e_n), l]$

if $\mathcal{C}(\exists v_{\mathcal{C}}.)\ \mathcal{D}(\exists v v_{\mathcal{D}}, v v'_{\mathcal{D}}\ (v v_{\mathcal{D}} \neq v v'_{\mathcal{D}})\ \in Dom(a_{\mathcal{D}}).)\ \forall \langle \mathcal{D}(v_{\mathcal{D}})(,)\rangle \in \{\langle \mathcal{D}(v v_{\mathcal{D}})(,)\rangle, \langle \mathcal{D}(v v_{\mathcal{D}})(,)\rangle\}.$
$\mathcal{B}(\exists v'_{\mathcal{B}}.)\ \exists 2(r \in Dom(f_b)).\ \mathcal{A}(\exists v_{\mathcal{A}}.)\ \mathcal{B}(\exists v_{\mathcal{B}}\ (\neq v'_{\mathcal{B}}).)$
$f_b(v_1, \ldots, v_n) = r$
$\mathcal{A}(\mathbf{cta}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,),\ \mathcal{B}(\mathbf{cpa}[e_{\mathcal{B}}, l_{\mathcal{B}}])(,),\ \mathcal{C}(\mathbf{can}[e_{\mathcal{C}},\ v_{\mathcal{C}}])(,) \rightarrow \mathbf{cpa}[_{l} f_b(e_1, \ldots, e_n), l]$

if $\mathcal{C}(\exists v_{\mathcal{C}}.)\ \mathcal{B}(\exists v'_{\mathcal{B}}.)\ \forall r \in Dom(f_b).\ \mathcal{A}(\exists v_{\mathcal{A}}.)\ \mathcal{B}(\exists v_{\mathcal{B}}\ (\neq v'_{\mathcal{B}}).)$
$f_b(v_1, \ldots, v_n) = r$
$\mathcal{A}(\mathbf{ota}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,),\ \mathcal{B}(\mathbf{opa}[e_{\mathcal{B}}, l_{\mathcal{B}}])(,) \rightarrow \mathbf{ota}[_{l} f_b(e_1, \ldots, e_n), l]$

if $\mathcal{C}(\exists v_{\mathcal{C}}.)\ \mathcal{B}(\exists v'_{\mathcal{B}}.)\ \exists 2(r \in Dom(f_b)).\ \mathcal{A}(\exists v_{\mathcal{A}}.)\ \mathcal{B}(\exists v_{\mathcal{B}}\ (\neq v'_{\mathcal{B}}).)$
$f_b(v_1, \ldots, v_n) = r$
$\mathcal{A}(\mathbf{ota}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,),\ \mathcal{B}(\mathbf{opa}[e_{\mathcal{B}}, l_{\mathcal{B}}])(,) \rightarrow \mathbf{opa}[_{l} f_b(e_1, \ldots, e_n), l]$

2, metarules for **can**
if $\mathcal{C}(\exists v_{\mathcal{C}}.)$
$\{\mathcal{D}(\exists v_{\mathcal{D}}.\ \in Dom(a_{\mathcal{D}}).)\ \exists r.\ \mathcal{A}(\forall v_{\mathcal{A}}.)\ \mathcal{B}(\forall v_{\mathcal{B}}\ (\neq v'_{\mathcal{B}}).)$
$f_b(v_1, \ldots, v_n) = r\}\ \wedge$
$\{\mathcal{D}(\exists v_{\mathcal{D}}.\ \in Dom(a_{\mathcal{D}}).)\ \exists 2(r \in Dom(f_b)).\ \mathcal{A}(\exists v_{\mathcal{A}}.)\ \mathcal{B}(\exists v_{\mathcal{B}}\ (\neq v'_{\mathcal{B}}).)$
$f_b(v_1, \ldots, v_n) = r\}$
$\mathcal{A}(\mathbf{cta}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,),\ \mathcal{B}(\mathbf{cpa}[e_{\mathcal{B}}, l_{\mathcal{B}}])(,),\ \mathcal{C}(\mathbf{can}[e_{\mathcal{C}},\ v_{\mathcal{C}}])(,) \rightarrow \mathbf{can}[_{l} f_b(e_1, \ldots, e_n),\ r]$

3, metarules for inferability
3.1, inferability on returned values
if $\mathcal{B}(\exists v_{\mathcal{B}}.)\ \mathcal{C}(\exists v v_{\mathcal{C}}, v v'_{\mathcal{C}}\ (v v_{\mathcal{C}} \neq v v'_{\mathcal{C}})\ \in Dom(a_{\mathcal{C}}).)\ \forall \langle \mathcal{C}(v_{\mathcal{C}})(,)\rangle \in \{\langle \mathcal{C}(v v_{\mathcal{C}})(,)\rangle, \langle \mathcal{C}(v v_{\mathcal{C}})(,)\rangle\}.\ \mathcal{A}(\exists v_{\mathcal{A}}.)$
$\exists r \in Dom(f_b).$
$\exists Q1$ s.t. $Q1$ is complete subset of $\mathcal{Q}1(Dom(e_{\mathcal{Q}1}))(\times).$
$\exists Q2$ s.t. $Q2$ is complete subset of $\mathcal{Q}2(Dom(e_{\mathcal{Q}2}))(\times).\ \ldots$
$\Pi(f_b(e_1, \ldots, e_n))($
$[\langle e_1, \ldots, e_n, f_b(e_1, \ldots, e_n)\rangle\ \in\ \{\langle w_1, \ldots, w_n, rr\rangle\ |\ f_b(w_1, \ldots, w_n) = rr)\}]\ \bowtie$
$\mathcal{P}([\langle e_{\mathcal{P}}\rangle\ \in\ \{v_{\mathcal{P}}\}])(\bowtie)$
$[\langle \mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle\ \in\ Q1]\ \bowtie\ [\langle \mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle\ \in\ Q2]\ \bowtie\ \ldots)$
$=\ [\langle f_b(e_1, \ldots, e_n)\rangle\ \in\ \{r\}]$
$\mathcal{A}(\mathbf{cpa}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,),\ \mathcal{B}(\mathbf{can}[e_{\mathcal{B}},\ v_{\mathcal{B}}])(,),\ \mathcal{P}(\mathbf{cti}[e_{\mathcal{P}}, S_{\mathcal{P}}])(,),$
$\mathbf{cpi}[\langle \mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle, SS_1],\ \mathbf{cpi}[\langle \mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle, SS_2],\ \ldots$

$$\rightarrow \mathbf{cti}[_l f_b(e_1,\ldots,e_n),\{\langle l,+\rangle\}] \qquad (\mathcal{P}(S_\mathcal{P} \neq \{\langle l,-\rangle\})(,),\ (SS_1 \neq \{\langle l,-\rangle\})$$

if $\mathcal{B}(\exists v_\mathcal{B}.)\ \mathcal{C}(\exists vv_\mathcal{C}, vv'_\mathcal{C}\ (vv_\mathcal{C} \neq vv'_\mathcal{C}.)\ \in Dom(a_\mathcal{C}).)\ \forall\langle\mathcal{C}(v_\mathcal{C})(,)\rangle \in \{\langle\mathcal{C}(vv_\mathcal{C})(,)\rangle, \langle\mathcal{C}(vv_\mathcal{C})(,)\rangle\}.\ \mathcal{A}(\exists v_\mathcal{A}.)$
$\exists R \subset Dom(f_b)\mathcal{R}(\times Dom(e_\mathcal{R}).$
$\exists Q1$ s.t. $Q1$ is complete subset of $\mathcal{Q}1(Dom(e_{\mathcal{Q}1}))(\times)$
$\exists Q2$ s.t. $Q2$ is complete subset of $\mathcal{Q}2(Dom(e_{\mathcal{Q}2}))(\times)\ \ldots$
$\Pi(f_b(e_1,\ldots,e_n),\mathcal{R}(e_\mathcal{R})(,))($
$[\langle e_1,\ldots,e_n,f_b(e_1,\ldots,e_n)\rangle\ \in\ \{\langle w_1,\ldots,w_n,rr\rangle \mid f_b(w_1,\ldots,w_n) = rr)\}]\ \bowtie$
$\mathcal{P}([\langle e_\mathcal{P}\rangle\ \in\ \{v_\mathcal{P}\}])(\bowtie)$
$[\langle\mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle\ \in\ Q1]\ \bowtie\ [\langle\mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle\ \in\ Q2]\ \bowtie\ \ldots)$
$=\ [\langle f_b(e_1,\ldots,e_n),\mathcal{R}(e_\mathcal{R})(,)\rangle\ \in\ R]$
$\quad \mathcal{A}(\mathbf{cpa}[e_\mathcal{A}, l_\mathcal{A}])(,),\ \mathcal{B}(\mathbf{can}[e_\mathcal{B},\ v_\mathcal{B}])(,),\ \mathcal{P}(\mathbf{cti}[e_\mathcal{P}, S_\mathcal{P}])(,),$
$\quad \mathbf{cpi}[\langle\mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle, SS_1],\ \mathbf{cpi}[\langle\mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle, SS_2],\ \ldots$
$\quad\quad \rightarrow \mathbf{cpi}[\langle_l f_b(e_1,\ldots,e_n),\mathcal{R}(e_\mathcal{R},)\rangle,\{\langle l,+\rangle\}] \qquad (\mathcal{P}(S_\mathcal{P} \neq \{\langle l,-\rangle\})(,),\ (SS_1 \neq \{\langle l,-\rangle\})$

if $\exists r \in Dom(f_b).\ \mathcal{P}(\exists v_\mathcal{P}.)$
$\exists Q1$ s.t. $Q1$ is complete subset of $\mathcal{Q}1(Dom(e_{\mathcal{Q}1}))(\times)$
$\exists Q2$ s.t. $Q2$ is complete subset of $\mathcal{Q}2(Dom(e_{\mathcal{Q}2}))(\times)\ \ldots$
$\Pi(f_b(e_1,\ldots,e_n))($
$[\langle e_1,\ldots,e_n,f_b(e_1,\ldots,e_n)\rangle\ \in\ \{\langle w_1,\ldots,w_n,rr\rangle \mid f_b(w_1,\ldots,w_n) = rr)\}]\ \bowtie$
$\mathcal{P}([\langle e_\mathcal{P}\rangle\ \in\ \{v_\mathcal{P}\}])(\bowtie)$
$[\langle\mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle\ \in\ Q1]\ \bowtie\ [\langle\mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle\ \in\ Q2]\ \bowtie\ \ldots)$
$=\ [\langle f_b(e_1,\ldots,e_n)\rangle\ \in\ \{r\}]$
$\quad \mathcal{P}(\mathbf{oti}[e_\mathcal{P}, S_\mathcal{P}])(,),$
$\quad \mathbf{opi}[\langle\mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle, SS_1],\ \mathbf{opi}[\langle\mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle, SS_2],\ \ldots$
$\quad\quad \rightarrow \mathbf{oti}[_l f_b(e_1,\ldots,e_n), l, +] \qquad (\mathcal{P}(S_\mathcal{P} \neq \{\langle l,-\rangle\})(,),\ (SS_1 \neq \{\langle l,-\rangle\})$

if $\exists R \subset Dom(f_b)\mathcal{R}(\times Dom(e_\mathcal{R}).\ \mathcal{P}(\exists v_\mathcal{P}.)$
$\exists Q1$ s.t. $Q1$ is complete subset of $\mathcal{Q}1(Dom(e_{\mathcal{Q}1}))(\times)$
$\exists Q2$ s.t. $Q2$ is complete subset of $\mathcal{Q}2(Dom(e_{\mathcal{Q}2}))(\times)\ \ldots$
$\Pi(f_b(e_1,\ldots,e_n),\mathcal{R}(e_\mathcal{R})(,))($
$[\langle e_1,\ldots,e_n,f_b(e_1,\ldots,e_n)\rangle\ \in\ \{\langle w_1,\ldots,w_n,rr\rangle \mid f_b(w_1,\ldots,w_n) = rr)\}]\ \bowtie$
$\mathcal{P}([\langle e_\mathcal{P}\rangle\ \in\ \{v_\mathcal{P}\}])(\bowtie)$
$[\langle\mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle\ \in\ Q1]\ \bowtie\ [\langle\mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle\ \in\ Q2]\ \bowtie\ \ldots)$
$=\ [\langle f_b(e_1,\ldots,e_n),\mathcal{R}(e_\mathcal{R})(,)\rangle\ \in\ R]$
$\quad \mathcal{P}(\mathbf{oti}[e_\mathcal{P}, S_\mathcal{P}])(,),$

$\mathbf{opi}[\langle \mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle, SS_1],\ \mathbf{opi}[\langle \mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle, SS_2],\ \ldots$

$\quad \rightarrow \mathbf{opi}[\langle_l f_b(e_1,\ldots,e_n), \mathcal{R}(e_{\mathcal{R}}),)\rangle, \{\langle l,+\rangle\}] \qquad (\mathcal{P}(S_{\mathcal{P}} \neq \{\langle l,-\rangle\})(,),\ (SS_1 \neq \{\langle l,-\rangle\})$

3.2, inferability on arguments with total inferability on returned values

if $\mathcal{B}(\exists v_{\mathcal{B}}.)\ \mathcal{C}(\exists vv_{\mathcal{C}}, vv'_{\mathcal{C}}\ (vv_{\mathcal{C}} \neq vv'_{\mathcal{C}})\ \in Dom(a_{\mathcal{C}}).)\ \forall\langle \mathcal{C}(v_{\mathcal{C}})(,)\rangle \in \{\langle \mathcal{C}(vv_{\mathcal{C}})(,)\rangle, \langle \mathcal{C}(vv_{\mathcal{C}})(,)\rangle\}.$

$\bigcup_{\mathcal{A}(v_{\mathcal{A}} \in Dom(a_{\mathcal{A}}))(,)}\{w_i \in Dom(a_i) | \exists S.$

$\exists Q1$ s.t. $Q1$ is complete subset of $\mathcal{Q}1(Dom(e_{\mathcal{Q}1}))(\times)$

$\exists Q2$ s.t. $Q2$ is complete subset of $\mathcal{Q}2(Dom(e_{\mathcal{Q}2}))(\times)\ \ldots$

$\Pi(e_i)($

$[\langle e_1,\ldots,e_n, f_b(e_1,\ldots,e_n)\rangle\ \in\ \{\langle w_1,\ldots,w_n, rr\rangle \mid f_b(w_1,\ldots,w_n) = rr\}]\ \bowtie$

$[f_b(e_1,\ldots,e_n)\ \in\ \{f_b(v_1,\ldots,v_n)\}] \bowtie\ \mathcal{P}([\langle e_{\mathcal{P}}\rangle\ \in\ \{v_{\mathcal{P}}\}])(\bowtie)$

$[\langle \mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle\ \in\ Q1]\ \bowtie\ [\langle \mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle\ \in\ Q2]\ \bowtie\ \ldots)$

$=\ [e_i\ \in\ S]\ \wedge w_i \notin S\} = Dom(a_i) \setminus \{v_i\}$

$\quad \mathcal{A}(\mathbf{cpa}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,),\ \mathcal{B}(\mathbf{can}[e_{\mathcal{B}},\ v_{\mathcal{B}}])(,),\ \mathcal{P}(\mathbf{cti}[e_{\mathcal{P}}, S_{\mathcal{P}}])(,),$

$\quad \mathbf{cpi}[\langle \mathcal{Q}1(e_{\mathcal{Q}}1)(,)\rangle, SS_1],\ \mathbf{cpi}[\langle \mathcal{Q}2(e_{\mathcal{Q}}2)(,)\rangle, SS_2],\ \mathbf{cti}[_l f_b(e_1,\ldots,e_n), S_0]$

$\qquad \rightarrow \mathbf{cti}[e_i, \{\langle l,-\rangle\}] \qquad (S_0 \neq \langle l,+\rangle)$

if $\mathcal{B}(\exists v_{\mathcal{B}}.)\ \mathcal{C}(\exists vv_{\mathcal{C}}, vv'_{\mathcal{C}}\ (vv_{\mathcal{C}} \neq vv'_{\mathcal{C}}) \in Dom(a_{\mathcal{C}}).)\ \forall\langle \mathcal{C}(v_{\mathcal{C}})(,)\rangle \in \{\langle \mathcal{C}(vv_{\mathcal{C}})(,)\rangle, \langle \mathcal{C}(vv_{\mathcal{C}})(,)\rangle\}.\ \mathcal{A}(\exists v_{\mathcal{A}}.)$

$\exists R \subset \mathcal{R}(Dom(a_{\mathcal{R}}))(\times).$

$\exists Q1$ s.t. $Q1$ is complete subset of $\mathcal{Q}1(Dom(e_{\mathcal{Q}1}))(\times).$

$\exists Q2$ s.t. $Q2$ is complete subset of $\mathcal{Q}2(Dom(e_{\mathcal{Q}2}))(\times).\ \ldots$

$\Pi(\mathcal{R}(e_{\mathcal{R}})(,))($

$[\langle e_1,\ldots,e_n, f_b(e_1,\ldots,e_n)\rangle\ \in\ \{\langle w_1,\ldots,w_n, rr\rangle \mid f_b(w_1,\ldots,w_n) = rr)\}]\ \bowtie$

$[f_b(e_1,\ldots,e_n)\ \in\ \{f_b(v_1,\ldots,v_n)\}] \bowtie\ \mathcal{P}([\langle e_{\mathcal{P}}\rangle\ \in\ \{v_{\mathcal{P}}\}])(\bowtie)$

$[\langle \mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle\ \in\ Q1]\ \bowtie\ [\langle \mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle\ \in\ Q2]\ \bowtie\ \ldots)v$

$=\ [\langle \mathcal{R}(e_{\mathcal{R}})(,)\rangle\ \in\ R]$

$\quad \mathcal{A}(\mathbf{cpa}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,),\ \mathcal{B}(\mathbf{can}[e_{\mathcal{B}},\ v_{\mathcal{B}}])(,),\ \mathcal{P}(\mathbf{cti}[e_{\mathcal{P}}, S_{\mathcal{P}}])(,),$

$\quad \mathbf{cpi}[\langle \mathcal{Q}1(e_{\mathcal{Q}}1)(,)\rangle, SS_1],\ \mathbf{cpi}[\langle \mathcal{Q}2(e_{\mathcal{Q}}2)(,)\rangle, SS_2],\ \mathbf{cti}[_l f_b(e_1,\ldots,e_n), S_0]$

$\qquad \rightarrow \mathbf{cpi}[\langle \mathcal{R}(e_{\mathcal{R}})(,)\rangle, \{\langle l,-\rangle\}] \qquad (S_0 \neq \langle l,+\rangle)$

if $\mathcal{B}(\exists v_{\mathcal{B}}.)\ \bigcup_{\mathcal{A}(v_{\mathcal{A}} \in Dom(a_{\mathcal{A}}))(,)}\{w_i \in Dom(a_i) | \exists S.$

$\exists Q1$ s.t. $Q1$ is complete subset of $\mathcal{Q}1(Dom(e_{\mathcal{Q}1}))(\times).$

$\exists Q2$ s.t. $Q2$ is complete subset of $\mathcal{Q}2(Dom(e_{\mathcal{Q}2}))(\times).\ \ldots$

$\Pi(e_i)($

$[\langle e_1,\ldots,e_n, f_b(e_1,\ldots,e_n)\rangle\ \in\ \{\langle w_1,\ldots,w_n, rr\rangle \mid f_b(w_1,\ldots,w_n) = rr\}]\ \bowtie$

$[f_b(e_1,\ldots,e_n)\ \in\ \{f_b(v_1,\ldots,v_n)\}] \bowtie\ \mathcal{P}([\langle e_{\mathcal{P}}\rangle\ \in\ \{v_{\mathcal{P}}\}])(\bowtie)$

$[\langle \mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle \in Q1] \bowtie [\langle \mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle \in Q2] \bowtie \dots)$

$= [\langle f_b(e_1,\dots,e_n)\rangle \in S] \wedge w_i \notin S\} = Dom(a_i) \setminus \{v_i\}$

$\mathcal{A}(\mathbf{opa}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,),\ \mathcal{P}(\mathbf{oti}[e_{\mathcal{P}}, S_{\mathcal{P}}])(,),$

$\mathbf{opi}[\langle \mathcal{Q}1(e_{\mathcal{Q}}1)(,)\rangle, SS_1],\ \mathbf{opi}[\langle \mathcal{Q}2(e_{\mathcal{Q}}2)(,)\rangle, SS_2],\ \mathbf{oti}[_l f_b(e_1,\dots,e_n), S_0]$

$\rightarrow \mathbf{oti}[e_i, \{\langle l, -\rangle\}] \qquad (S_0 \neq \langle l, +\rangle)$

if $\exists v_1 \in Dom(a_1), \dots, v_n \in Dom(a_n)$.

$\exists R \subset \mathcal{R}(Dom(a_{\mathcal{R}}))(\times)$.

$\exists Q1$ s.t. $Q1$ is complete subset of $\mathcal{Q}1(Dom(e_{\mathcal{Q}1}))(\times)$.

$\exists Q2$ s.t. $Q2$ is complete subset of $\mathcal{Q}2(Dom(e_{\mathcal{Q}2}))(\times)$. ...

$\Pi(\mathcal{R}(e_{\mathcal{R}})(,))($

$[\langle e_1,\dots,e_n, f_b(e_1,\dots,e_n)\rangle \in \{\langle w_1,\dots,w_n, rr\rangle \mid f_b(w_1,\dots,w_n) = rr\}] \bowtie$

$[f_b(e_1,\dots,e_n) \in \{f_b(v_1,\dots,v_n)\}] \bowtie \mathcal{P}([\langle e_{\mathcal{P}}\rangle \in \{v_{\mathcal{P}}\}])(\bowtie)$

$[\langle \mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle \in Q1] \bowtie [\langle \mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle \in Q2] \bowtie \dots)$

$= [\langle \mathcal{R}(e_{\mathcal{R}})(,)\rangle \in R]$

$\mathcal{A}(\mathbf{opa}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,),\ \mathcal{P}(\mathbf{oti}[e_{\mathcal{P}}, S_{\mathcal{P}}])(,),$

$\mathbf{opi}[\langle \mathcal{Q}1(e_{\mathcal{Q}}1)(,)\rangle, SS_1],\ \mathbf{opi}[\langle \mathcal{Q}2(e_{\mathcal{Q}}2)(,)\rangle, SS_2],\ \mathbf{oti}[_l f_b(e_1,\dots,e_n), S_0]$

$\rightarrow \mathbf{opi}[\langle \mathcal{R}(e_{\mathcal{R}})(,)\rangle, \{\langle l, -\rangle\}] \qquad (S_0 \neq \langle l, +\rangle)$

3.2, inferability on arguments with partial inferability on returned values

if $\mathcal{B}(\exists v_{\mathcal{B}}.)\ \mathcal{C}(\exists vv_{\mathcal{C}}, vv'_{\mathcal{C}}\ (vv_{\mathcal{C}} \neq vv'_{\mathcal{C}}) \in Dom(a_{\mathcal{C}}).)\ \forall \langle \mathcal{C}(v_{\mathcal{C}})(,)\rangle \in \{\langle \mathcal{C}(vv_{\mathcal{C}})(,)\rangle, \langle \mathcal{C}(vv_{\mathcal{C}})(,)\rangle\}$.

$\bigcup_{\mathcal{A}(v_{\mathcal{A}} \in Dom(a_{\mathcal{A}}))(,)} \{w_i \in Dom(a_i) | \exists S.\ \exists O \subset Dom(f_b)$

$\exists Q1$ s.t. $Q1$ is complete subset of $\mathcal{Q}1(Dom(e_{\mathcal{Q}1}))(\times)$.

$\exists Q2$ s.t. $Q2$ is complete subset of $\mathcal{Q}2(Dom(e_{\mathcal{Q}2}))(\times)$. ...

$\Pi(e_i)($

$[\langle e_1,\dots,e_n, f_b(e_1,\dots,e_n)\rangle \in \{\langle w_1,\dots,w_n, rr\rangle \mid f_b(w_1,\dots,w_n) = rr\}] \bowtie$

$[f_b(e_1,\dots,e_n) \in O] \bowtie \mathcal{P}([\langle e_{\mathcal{P}}\rangle \in \{v_{\mathcal{P}}\}])(\bowtie)$

$[\langle \mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle \in Q1] \bowtie [\langle \mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle \in Q2] \bowtie \dots)$

$= [e_i \in S] \wedge w_i \notin S\} = Dom(a_i) \setminus \{v_i\}$

$\mathcal{A}(\mathbf{cpa}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,),\ \mathcal{B}(\mathbf{can}[e_{\mathcal{B}},\ v_{\mathcal{B}}])(,),\ \mathcal{P}(\mathbf{cti}[e_{\mathcal{P}}, S_{\mathcal{P}}])(,),$

$\mathbf{cpi}[\langle \mathcal{Q}1(e_{\mathcal{Q}}1)(,)\rangle, SS_1],\ \mathbf{cpi}[\langle \mathcal{Q}2(e_{\mathcal{Q}}2)(,)\rangle, SS_2],\ \mathbf{cpi}[_l f_b(e_1,\dots,e_n), S_0]$

$\rightarrow \mathbf{cti}[e_i, \{\langle l, -\rangle\}] \qquad (S_0 \neq \langle l, +\rangle)$

if $\mathcal{B}(\exists v_{\mathcal{B}}.)\ \mathcal{C}(\exists vv_{\mathcal{C}}, vv'_{\mathcal{C}}\ (vv_{\mathcal{C}} \neq vv'_{\mathcal{C}}) \in Dom(a_{\mathcal{C}}).)\ \forall \langle \mathcal{C}(v_{\mathcal{C}})(,)\rangle \in \{\langle \mathcal{C}(vv_{\mathcal{C}})(,)\rangle, \langle \mathcal{C}(vv_{\mathcal{C}})(,)\rangle\}.\ \mathcal{A}(\exists v_{\mathcal{A}}.)$

$\exists O \subset Dom(f_b)\ \exists R \subset \mathcal{R}(Dom(a_{\mathcal{R}}))(\times)$.

$\exists Q1$ s.t. $Q1$ is complete subset of $\mathcal{Q}1(Dom(e_{\mathcal{Q}1}))(\times)$.

$\exists Q2$ s.t. $Q2$ is complete subset of $\mathcal{Q}2(Dom(e_{\mathcal{Q}2}))(\times)$. ...

$\Pi(\mathcal{R}(e_{\mathcal{R}})(,))($

$[\langle e_1,\ldots,e_n, f_b(e_1,\ldots,e_n)\rangle \in \{\langle w_1,\ldots,w_n,rr\rangle \mid f_b(w_1,\ldots,w_n)=rr\}] \bowtie$

$[f_b(e_1,\ldots,e_n) \in O] \bowtie \mathcal{P}([\langle e_{\mathcal{P}}\rangle \in \{v_{\mathcal{P}}\}])(\bowtie)$

$[\langle \mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle \in Q1] \bowtie [\langle \mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle \in Q2] \bowtie \ldots)$

$=$ $[\langle \mathcal{R}(e_{\mathcal{R}})(,)\rangle \in R]$

$\mathcal{A}(\mathbf{cpa}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,)$, $\mathcal{B}(\mathbf{can}[e_{\mathcal{B}}, v_{\mathcal{B}}])(,)$, $\mathcal{P}(\mathbf{cti}[e_{\mathcal{P}}, S_{\mathcal{P}}])(,)$,

$\mathbf{cpi}[\langle \mathcal{Q}1(e_{\mathcal{Q}}1)(,)\rangle, SS_1]$, $\mathbf{cpi}[\langle \mathcal{Q}2(e_{\mathcal{Q}}2)(,)\rangle, SS_2]$, $\mathbf{cpi}[_l f_b(e_1,\ldots,e_n), S_0]$

$\rightarrow \mathbf{cpi}[\langle \mathcal{R}(e_{\mathcal{R}})(,)\rangle, \{\langle l,-\rangle\}] \qquad (S_0 \neq \langle l,+\rangle)$

if $\mathcal{B}(\exists v_{\mathcal{B}}.) \bigcup_{\mathcal{A}(v_{\mathcal{A}} \in Dom(a_{\mathcal{A}}))(,)} \{w_i \in Dom(a_i) | \exists S.\ \exists O \subset Dom(f_b)\ \exists Q1$ s.t. $Q1$ is complete subset of $\mathcal{Q}1(Dom(e_{\mathcal{Q}1}))(\times)$.

$\exists Q2$ s.t. $Q2$ is complete subset of $\mathcal{Q}2(Dom(e_{\mathcal{Q}2}))(\times)$. ...

$\Pi(e_i)($

$[\langle e_1,\ldots,e_n, f_b(e_1,\ldots,e_n)\rangle \in \{\langle w_1,\ldots,w_n,rr\rangle \mid f_b(w_1,\ldots,w_n)=rr\}] \bowtie$

$[f_b(e_1,\ldots,e_n) \in O] \bowtie \mathcal{P}([\langle e_{\mathcal{P}}\rangle \in \{v_{\mathcal{P}}\}])(\bowtie)$

$[\langle \mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle \in Q1] \bowtie [\langle \mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle \in Q2] \bowtie \ldots)$

$=$ $[\langle f_b(e_1,\ldots,e_n)\rangle \in S] \wedge w_i \notin S\} = Dom(a_i) \setminus \{v_i\}$

$\mathcal{A}(\mathbf{opa}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,)$, $\mathcal{P}(\mathbf{oti}[e_{\mathcal{P}}, S_{\mathcal{P}}])(,)$,

$\mathbf{opi}[\langle \mathcal{Q}1(e_{\mathcal{Q}}1)(,)\rangle, SS_1]$, $\mathbf{opi}[\langle \mathcal{Q}2(e_{\mathcal{Q}}2)(,)\rangle, SS_2]$, $\mathbf{oti}[_l f_b(e_1,\ldots,e_n), S_0]$

$\rightarrow \mathbf{oti}[e_i, \{\langle l,-\rangle\}] \qquad (S_0 \neq \langle l,+\rangle)$

if $\exists v_1 \in Dom(a_1),\ldots,v_n \in Dom(a_n)$.

$\exists O \subset Dom(f_b)\ \exists R \subset \mathcal{R}(Dom(a_{\mathcal{R}}))(\times)$.

$\exists Q1$ s.t. $Q1$ is complete subset of $\mathcal{Q}1(Dom(e_{\mathcal{Q}1}))(\times)$.

$\exists Q2$ s.t. $Q2$ is complete subset of $\mathcal{Q}2(Dom(e_{\mathcal{Q}2}))(\times)$. ...

$\Pi(\mathcal{R}(e_{\mathcal{R}})(,))($

$[\langle e_1,\ldots,e_n, f_b(e_1,\ldots,e_n)\rangle \in \{\langle w_1,\ldots,w_n,rr\rangle \mid f_b(w_1,\ldots,w_n)=rr\}] \bowtie$

$[f_b(e_1,\ldots,e_n) \in O] \bowtie \mathcal{P}([\langle e_{\mathcal{P}}\rangle \in \{v_{\mathcal{P}}\}])(\bowtie)$

$[\langle \mathcal{Q}1(e_{\mathcal{Q}1})(,)\rangle \in Q1] \bowtie [\langle \mathcal{Q}2(e_{\mathcal{Q}2})(,)\rangle \in Q2] \bowtie \ldots)$

$=$ $[\langle \mathcal{R}(e_{\mathcal{R}})(,)\rangle \in R]$

$\mathcal{A}(\mathbf{opa}[e_{\mathcal{A}}, l_{\mathcal{A}}])(,)$, $\mathcal{P}(\mathbf{oti}[e_{\mathcal{P}}, S_{\mathcal{P}}])(,)$,

$\mathbf{opi}[\langle \mathcal{Q}1(e_{\mathcal{Q}}1)(,)\rangle, SS_1]$, $\mathbf{opi}[\langle \mathcal{Q}2(e_{\mathcal{Q}}2)(,)\rangle, SS_2]$, $\mathbf{oti}[_l f_b(e_1,\ldots,e_n), S_0]$

$\rightarrow \mathbf{opi}[\langle \mathcal{R}(e_{\mathcal{R}})(,)\rangle, \{\langle l,-\rangle\}] \qquad (S_0 \neq \langle l,+\rangle)$

The conditions on rules may appear to be complex but they are just composed of relatively small number of repeatedly used parts. In the rules deducing some certain capability, conditions are started with quantifiers corresponding $\forall D$, that is, "for any initial database state". In those quantifiers, for arguments on which the user does not have any alterability or **can**, we use "$\exists 2(\langle \mathcal{A}(v_{\mathcal{A}})(,)\rangle \in \mathcal{A}(Dom(a_{\mathcal{A}}))(\times))$.", i.e. if $A = \{i, j\}$, "$\exists 2(\langle v_i, v_j\rangle \in Dom(a_i) \times Dom(a_j))$." instead of "$\forall v_i.\ \forall v_j.$" because in the worst case e_i and e_j may take only two combinations of values for all considerable initial database states. For arguments on which the user have total alterability, we use "$\exists v_i$" and place it after $\forall$ for arguments without alterability because the user can choose any appropriate values for those arguments. We also use "$\exists v_i$" but we place it before $\forall$ for arguments without alterability because there can be only one value for which **can** holds.

In the rules deducing total alterability, we use "$\forall r \in Dom(f_b).\ \exists v_i.$" for arguments with total alterability, and "$\exists v_i'.\ \forall r.\ \exists v_i (\neq v_i').$" for arguments with partial alterability because partial alterability means there is at least one value v_i' which the user cannot let the expression take. For rules deducing partial alterability, we use "$\exists 2(r \in Dom(f_b))$." instead of "$\forall r \in Dom(f_b)$." because partial alterability means there are at least two values that the user can let the expression take.

The rules for inferability are examining whether the set S in the term $[e \in S]$ deduced by $\mathcal{I}(E)$ can be a singleton set or not. The conditions on the rules deducing certain inferability is composed of two parts. First, quantifiers representing $\forall D$ are placed. Those quantifiers bound variables $v_1, \ldots, v_n$ to true values of $e_1, \ldots, e_n$. The second part corresponds to the inference by the user. For arguments with partial inferability, we use "$\exists Q$ which is a complete subset of ...". However, when one has partial inferability on $\langle e_1, \ldots, e_n\rangle$, he may infer $[\langle e_1, \ldots, e_n\rangle \in S]$ with S that is NOT a complete subset. For example, $\Pi(e_i)S \subset Dom(e_i)$ may hold. In that case, $\mathbf{cpi}[e_i, SS]$ must also be deduced, and S can be expressed as a join of $S_i \subset Dom(e_i)$ and S' which is a complete subset of $Dom(e_1) \times \ldots \times Dom(e_n)$. Therefore, we can proceed analysis with assuming the user can infer only a complete subset of $Dom(e_1) \times \ldots \times Dom(e_n)$.

"$\bigcup\{\ldots\}$" corresponds to the repetition of inference on the same expression with changing some alterable arguments. We explain the effect of such repetition in Section 3.3.

In the rules deducing inferability, we do not distinguish total alterability and partial alterability. Of course, we can define a little less pessimistic rules by distinguish them. Distinction of them, however, does not make so much improvement while it makes rules more complicated. Therefore, we omit that distinction here.

As an example of rules based on basic functions, we show the rules on the basic function >=(x, y);

$$
\begin{aligned}
\mathbf{cpa}[\mathrm{x}] &\rightarrow \mathbf{cta}[>=(\mathrm{x},\ \mathrm{y})] \\
\mathbf{opa}[\mathrm{x}] &\rightarrow \mathbf{ota}[>=(\mathrm{x},\ \mathrm{y})] \\
\mathbf{cpi}[\mathrm{x}], \mathbf{cpi}[\mathrm{y}] &\rightarrow \mathbf{cti}[>=(\mathrm{x},\ \mathrm{y})] \\
\mathbf{opi}[\mathrm{x}], \mathbf{opi}[\mathrm{y}] &\rightarrow \mathbf{oti}[>=(\mathrm{x},\ \mathrm{y})] \\
\mathbf{cpi}[\mathrm{x}] &\rightarrow \mathbf{cpi}[\langle \mathrm{y},\ >=(\mathrm{x},\ \mathrm{y})\rangle] \\
\mathbf{opi}[\mathrm{x}] &\rightarrow \mathbf{opi}[\langle \mathrm{y},\ >=(\mathrm{x},\ \mathrm{y})\rangle]
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{cti}[x], \mathbf{cpa}[x], \mathbf{cti}[>=(x,\ y)] &\rightarrow \mathbf{cti}[y] \\
\mathbf{oti}[x], \mathbf{opa}[x], \mathbf{oti}[>=(x,\ y)] &\rightarrow \mathbf{oti}[y] \\
\mathbf{cpi}[x], \mathbf{cti}[>=(x,\ y)] &\rightarrow \mathbf{cpi}[y] \\
\mathbf{opi}[x], \mathbf{oti}[>=(x,\ y)] &\rightarrow \mathbf{opi}[y] \\
\mathbf{cti}[>=(x,\ y)] &\rightarrow \mathbf{cpi}[\langle x,\ y\rangle] \\
\mathbf{oti}[>=(x,\ y)] &\rightarrow \mathbf{opi}[\langle x,\ y\rangle]
\end{aligned}
$$

We can get more rules from the definition above, but we omit redundant ones, such as "**cpa**[x], **cpa**[y] → **cta**[>=(x, y)]", and those that can be got by replacing x and y, such as "**cpa**[y] → **cta**[>=(x, y)]". In the same way, the rules for ∗(x, y), multiplication on integers, are as follows;

$$
\begin{aligned}
\mathbf{cta}[x], \mathbf{cta}[y] &\rightarrow \mathbf{cta}[*(x,\ y)] \\
\mathbf{cta}[x], \mathbf{can}[y,\ 1] &\rightarrow \mathbf{cta}[*(x,\ y)] \\
\mathbf{ota}[x] &\rightarrow \mathbf{ota}[*(x,\ y)] \\
\mathbf{cpa}[x] &\rightarrow \mathbf{cpa}[*(x,\ y)] \\
\mathbf{opa}[x] &\rightarrow \mathbf{opa}[*(x,\ y)] \\
\mathbf{cti}[x], \mathbf{cti}[y] &\rightarrow \mathbf{cti}[*(x,\ y)] \\
\mathbf{cti}[x], \mathbf{cpa}[x] &\rightarrow \mathbf{cti}[*(x,\ y)] \\
\mathbf{cti}[x], \mathbf{can}[x,\ 0] &\rightarrow \mathbf{cti}[*(x,\ y)] \\
\mathbf{oti}[x] &\rightarrow \mathbf{oti}[*(x,\ y)] \\
\mathbf{cpi}[x] &\rightarrow \mathbf{cpi}[*(x,\ y)] \\
\mathbf{opi}[x] &\rightarrow \mathbf{opi}[*(x,\ y)] \\
\mathbf{cpi}[x] &\rightarrow \mathbf{cpi}[\langle y, *(x,\ y)\rangle] \\
\mathbf{opi}[x] &\rightarrow \mathbf{opi}[\langle y, *(x,\ y)\rangle] \\
\mathbf{cti}[x], \mathbf{cti}[*(x,\ y)] &\rightarrow \mathbf{cti}[y] \\
\mathbf{oti}[x], \mathbf{oti}[*(x,\ y)] &\rightarrow \mathbf{oti}[y] \\
\mathbf{cpi}[*(x,\ y)] &\rightarrow \mathbf{cpi}[y] \\
\mathbf{opi}[*(x,\ y)] &\rightarrow \mathbf{opi}[y] \\
\mathbf{cpi}[*(x,\ y)] &\rightarrow \mathbf{cpi}[\langle x,\ y\rangle] \\
\mathbf{opi}[*(x,\ y)] &\rightarrow \mathbf{opi}[\langle x,\ y\rangle]
\end{aligned}
$$

The rule **cti**[x], **can**[x, 0]→**cti**[∗(x, y)] holds because ∗(x, y) is 0 when x = 0.

Using $\mathcal{J}(F)$, we define the algorithm $\mathcal{A}(R)$ that determines whether the requirement R is satisfied or not as below;

Definition 12 $\mathcal{A}(R)$

Given $R = (u,\ f(x_1 : c_1^1 \ldots : c_1^{m_1}, \ldots, x_n : c_n^1 \ldots : c_n^{m_n}) : c_0^1 \ldots : c_0^{m_0},\ \mathcal{A}(R)$ *calculates the closure set of all inferable terms of* $\mathcal{J}(F)$ *where* F *is a set of all functions in the capability list of* u*. Then, if there exists some* ${}_if(_{l_1}e_1, \ldots {}_{,l_n}e_n) \in \mathcal{S}'(F)$ *for which all terms corresponding to capabilities specified in* R *are*

included in the closure set, $\mathcal{A}(R)$ determines that R is not satisfied. Otherwise $\mathcal{A}(R)$ determines that R is satisfied. □

$\mathcal{A}(R)$ can be proved to be sound;

Theorem 1 *Soundness of $\mathcal{A}(R)$*
If R is not satisfied, $\mathcal{A}(R)$ always determines that R is not satisfied.

(Proof outline) As for alterability and **can**, it is proved by induction on the structure of the subexpression. It is rather simple. As for inferability, it is proved by induction on the length of inference sequence of $\mathcal{I}(\mathcal{E})$ deducing $[e \in S]$. The soundness of the algorithm is essentially comes from the fact that "the rules for basic functions are defined so that the algorithm is sound". We briefly describe several points that are important in the proof.

1. Join operation in $\mathcal{I}(\mathcal{E})$ is associative and commutative. Therefore, the results that the user can get by join operations only depends on the set of terms used in join operations and independent from the order in which those terms are joined.

2. In $\mathcal{I}$, terms of the form $[\langle \ldots \rangle \in S]$ are deduced by (1) axioms (2) the rule deducing $[\langle e_1, e_2 \rangle \in \ldots]$ from equality (3) the rule corresponding to dependency among arguments and results of basic functions. Results got by (1) are simulated by axioms in $\mathcal{J}$. Results got by (2) are simulated by the rule $=[e_1,\ e_2] \rightarrow \mathbf{cpi}[\langle e_1, e_2 \rangle, \ldots]$ in $\mathcal{J}$. Results got by joining (1) and (2) are simulated by the rule $=[e_1,\ e_2]$, $\mathbf{cti}[e_1, S] \rightarrow \mathbf{cti}[e_2, S]$ and $=[e_1,\ e_2]\mathbf{cpi}[\langle \ldots, e_1, \ldots \rangle, S] \rightarrow \mathbf{cpi}[\langle \ldots, e_2, \ldots \rangle, S]$. Results got by (3) or by joining (3) and others are simulated by the rules for basic functions.

3. Two different occasional partial alterability can cause certain total alterability as explained before.

4. On the other hand, two different occasional partial inferability can never cause certain total inferability as explained before. They can cause just certain partial inferability and occasional total inferability.

5. The rule $\mathbf{opa}[e_1] \rightarrow \mathbf{ota}[\_r\_att(e_1)]$ holds because the order of $Dom(\_r\_att(e_1))$ may be smaller than $Dom(e_1)$. On the other hand, $\mathbf{cta}[e_1] \rightarrow \mathbf{cpa}[\_r\_att(e_1)]$ cannot hold because if every object in $Dom(e_1)$ has the same value in its attribute *att* in the initial database state, the result of $\_r\_att(e_1)$ is always the same value however he change the value of e_1.

■

3.4.2 An Example

We briefly explain one example of the analysis. Suppose the user u can directly invoke checkBudget(broker) and _w_budget(o, v), and requirement $(u, \_r\_salary(broker) : \mathbf{cti})$ is specified. In this situation, as explained

in Section 3.3, the user u can infer the exact value of the salary of each broker. Therefore, there is a security flaw. It can be detected in the following way. First, we unfold and number the code of checkBudget and _w_budget(o, v) as below;

checkBudget(broker)
  $_7$ >=($_2$_r_budget($_1$broker), $_6$*($_3$10, $_5$_r_salary($_4$broker)))

$_{10}$_w_budget($_8$o, $_9$v)

Then, $\mathcal{J}(\{$checkBudget(broker), _w_budget(o, v)$\})$ can deduce $\mathbf{cti}[^5$_r_salary(^{4}broker)] as shown below;

	$\rightarrow$	=[$_8$o, $_1$broker]	(axiom for =)
=[$_8$o, $_1$broker]	$\rightarrow$	=[$_9$v, $_2$_r_budget($_1$broker)]	(rule for =)
	$\rightarrow$	**cti**[$_9$v]	(axiom)
=[$_9$v, $_2$_r_budget($_1$broker)], **cti**[$_9$v]	$\rightarrow$	**cti**[$_2$_r_budget($_1$broker)]	(inferability based on =)
	$\rightarrow$	**cpa**[$_9$v]	(axiom)
=[$_8$o, $_1$broker], **cpa**[$_9$v]	$\rightarrow$	**cpa**[$_2$_r_budget($_1$broker)]	(alterability based on =)
	$\rightarrow$	**cti**[$_7$ >=(...)]	(axiom)
cti[$_2$_r_budget($_1$broker)], **cpa**[$_2$_r_budget($_1$broker)], **cti**[$_7$ >=(...)]			
	$\rightarrow$	**cti**[$_6$*($_3$10, $_5$_r_salary($_4$broker))]	(basic function)
	$\rightarrow$	**cti**[$_3$10]	(axiom)
cti[$_3$10], **cti**[$_6$*($_3$10, $_5$_r_salary($_4$broker))]			
	$\rightarrow$	**cti**[$_5$_r_salary($_4$broker)]	(basic function)

Thus, $\mathbf{cti}[^5$_r_salary(^{4}broker)] is deduced, and therefore, $\mathcal{A}(F)$ determines that (u,_r_salary(broker):**cti**) is not satisfied. This means the security flaw is successfully detected.

3.5 Contribution and Future Work

We define a framework for access control in the abstract operation granularity, and developed an mechanism that detects security flaws caused by functions not hiding primitive operations inside them. The most important contribution of this research is that in order to properly model the problem of that kind of security flaws, we introduced the notions of inferability on returned values and controllability on arguments, investigate their properties, and gave the formal semantics of the security requirement described in terms of those notions. We think that these notions are proper generalization of traditional read/write capability and can work as a basis for various researches on access control in the function granularity. In order to show usefulness of those notions we also developed a sound algorithm that determines whether a given requirement is satisfied or not.

Although we showed a static analysis algorithm which is sound and sufficiently practical, it is not necessarily one and only way to avoid security flaws. In fact, the algorithm we showed in this research is quite pessimistic instead of requiring relatively small amount of computation. More accurate analysis with more complex computation could be developed using existing techniques, such as several techniques proposed in the researches for the abstract interpretation of programs. Another alternative is to develop a mechanism to dynamically detect security flaws during execution of queries. Those are future issues.

We assume a rather simple data model in this development. In [DHP89, RBKW91, Spo89, GGF93], how various data modeling concepts, such as versions or inheritance, affect the authorization mechanisms is discussed. The integration of the techniques we show in this thesis and the mechanisms proposed in those researches is an issue for future researches. The function definition language we defined in this research is also quite simple language. Including more language features into it, such as recursion and polymorphism, is also an interesting issue for future researches.

In this thesis, we do not discuss about properties of aggregate functions on sets. Interesting studies on that topic has been shown in the context of statistical databases [KU77, Chi78, DDS79, DJL79, Bec80, CÖ82, LDS+90]. The result of these researches say, in short, that aggregate functions on a set of data almost always reveal the information on the individual elements of the set.

References

[AB91] Serge Abiteboul and Anthony Bonner. Objects and views. In *Proc. of ACM SIGMOD*, pages 238–247, Jun. 1991.

[ABD+89] Malcolm Atkisson, François Bancilhon, David DeWitt, Klaus R. Dittrich, David Maier, and Stanley B. Zdonik. The object-oriented database system manifesto. In *Proc. of Int. Conf. on Deductive and Object-Oriented Database Systems*, pages 40–57. Elsevier Science Publ., Dec. 1989.

[ABGO93] Antonio Albano, R. Bergamini, Giorgio Ghelli, and Renzo Orsini. An object data model with roles. In *Proc. of VLDB*, pages 39–51, Aug. 1993.

[ACO85] Antonio Albano, Luca Cardelli, and Renzo Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM TODS*, 10(2):230–260, Jun. 1985.

[ADG92] Rafiul Ahad, James Davis, and Stefan Gower. Supporting access control in an object-oriented database language. In *Proc. of EDBT*, volume 580 of *LNCS*, pages 184–200. Springer-Verlag, Mar. 1992.

[ALUW93] Serge Abiteboul, Georg Lausen, Heinz Uphoff, and Emmanuel Waller. Methods and rules. In *Proc. of ACM SIGMOD*, pages 32–41, May 1993.

[ANS75] ANSI/X3/SPARC Study group on data base management systems. Interim report. *FDT – Bulletin of ACM SIGMOD*, 7(2), 1975.

[BBKV88] François Bancilhon, Ted Briggs, Setrag Khoshafian, and Patrick Valduriez. FAD. a powerful and simple database language. In *Proc. of VLDB*, pages 97–105, Sep. 1988.

[BDK92] François Bancilhon, Claude Delobel, and Paris C. Kanellakis, editors. *Building an OODBMS, The Story of O_2*. Morgan Kaufmann, 1992.

[Bec80] Leland L. Beck. A security mechanism for statistical databases. *ACM TODS*, 5(3):316–338, Sep. 1980.

[Ber92] Elisa Bertino. Data hiding and security in object-oriented databases. In *Proc. of IEEE ICDE*, pages 338–347, Feb. 1992.

[BKSW91] Thierry Barsalou, Arthur M. Keller, Niki Siambela, and Gio Wiederhold. Updating relational databases through object-based views. In *Proc. of ACM SIGMOD*, pages 248–257, May 1991.

[BO91] Peter Buneman and Atsushi Ohori. A type system that reconciles classes and extents. In *Proc. of Int. Workshop on DBPL*, pages 191–202. Morgan Kaufmann, Aug. 1991.

[BO96] Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. *ACM TODS*, 1996. To appear in March issue.

[BTBO91] Val Breazu-Tannen, Peter Buneman, and Atsushi Ohori. Data structures and data types in object-oriented databases. *IEEE Data Engineering Bulletin, Special Issue on Theoretical Foundation of Object-Oriented Databases*, 14(2):23–27, Jun. 1991.

[Bur90] Rae K. Burns. Referential secrecy. In *Proc. of IEEE Symp. on Research in Security and Privacy*, pages 133–142, 1990.

[CDMB90] Richard C. H. Connor, Alan Dearle, Ronald Morrison, and Fred Brown. Existentially quantified types as a database viewing mechanism. In *Proc. of EDBT*, volume 416 of *LNCS*, pages 301–315. Springer-Verlag, Mar. 1990.

[Chi78] Francis Y. Chin. Security in statistical databases for queries with small counts. *ACM TODS*, 3(1):92–104, Mar. 1978.

[CÖ82] Francis Y. Chin and Gultekin Özsoyoglu. Auditing and inference control in statistical database. *IEEE Trans. on Soft. Eng.*, 8(6):574–582, Nov. 1982.

[Coh77] Ellis Cohen. Information transmission in computational systems. In *Proc. of ACM Symp. on OS Principles*, pages 133–139, Nov. 1977.

[DAH+87] Dorothy E. Denning, Selim G. Akl, Mark Heckman, Teresa F. Lunt, Matthew Morgenstern, Peter G. Neumann, and Roger R. Schell. Views for multilevel database security. *IEEE Trans. on Soft. Eng.*, 13(2):129–140, Feb. 1987.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *CACM*, 20(7):504–512, Jul. 1977.

[DDS79] Dorothy E. Denning, Peter J. Denning, and Mayer D. Schwartz. The tracker: A threat to statistical database security. *ACM TODS*, 4(1):76–96, Mar. 1979.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *CACM*, 19(5):236–243, May 1976.

[DHP89] Klaus R. Dittrich, Martin Härtig, and Heriber Pfefferle. Discretionary access control in structually object-oriented database systems. In *Database Security II: Status and Prospectus*, pages 105–121. Elsevier Science Publ., 1989.

[DJL79] David Dobkin, Anita K. Jones, and Richard J. Lipton. Secure databases: Protection against user influence. *ACM TODS*, 4(1):97–106, Mar. 1979.

[FSW81] Eduardo B. Fernández, Rita C. Summers, and Christopher Wood. *Database Security and Integrity*, chapter 5, pages 55–64. The Systems Programming Series. Addison-Wesley, 1981.

[GGF93] Nurith Gal-Oz, Ehud Gudes, and Eudardo B. Fernández. A model of methods authorization in object-oriented databases. In *Proc. of VLDB*, pages 52–61, Aug. 1993.

[GPZ88] Georg Gottlob, Paolo Paolini, and Roberto Zicari. Properties and update semantics of consistent views. *ACM TODS*, 13(4):486–524, Dec. 1988.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[HD92] Thomas Hinke and Harry S. Delugach. Aerie: An inference modeling and detection approach for databases. In *Database Security VI: Status and Prospects*, pages 179–194. IFIP WG 11.3, Aug. 1992.

[HO90] Brent Hailpern and Harold Ossher. Extending object to support multiple interfaces and access control. *IEEE Trans. on Soft. Eng.*, 16(11):1247–1257, Nov. 1990.

[HZ90] Sandra Heiler and Stanley B. Zdonik. Object views: Extending the vision. In *Proc. of IEEE ICDE*, pages 86–93, Feb. 1990.

[JM88] Laita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes. In *Proc. of ACM LFP*, pages 198–211, Jul. 1988.

[JS91] Sushil Jajodia and Ravi Sandhu. Toward a multilevel secure relational data model. In *Proc. of ACM SIGMOD*, pages 50–59, May 1991.

[KLW90] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. Tech. Rep. 90/14, State University of New York at Stony Brook, 1990.

[KU77] John B. Kam and Jeffrey D. Ullman. A model of statistiacl database and their security. *ACM TODS*, 2(1):1–10, Mar. 1977.

[LDS+90] Teresa F. Lunt, Dorothy E. Denning, Roger R. Schell, Mark Heckman, and William R. Shockley. The SeaView security model. *IEEE Trans. on Soft. Eng.*, 16(6):593–607, Jun. 1990.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, Dec. 1978.

[MJ88] Catherine Meadows and Sushil Jajodia. Ingetrity versus security in multi-level secure databases. In *Database Security II: Status and Prospects*, pages 89–101. IFIP WG 11.3, North-Holland, Oct. 1988.

[Mor87] Matthew Morgenstern. Security and inference in multilevel database and knowlege-base systems. In *Proc. of ACM SIGMOD*, pages 357–371, Dec. 1987.

[MSS88] Subhasish Mazumdar, David Stemple, and Tim Sheard. Resolving the tension between integrity and security using a theorem prover. In *Proc. of ACM SIGMOD*, pages 233–242, Sep. 1988.

[OB88] Atsushi Ohori and Peter Buneman. Type inference in a database programming language. In *Proc. of ACM LFP*, pages 174–183, Jul. 1988.

[OB89] Atushi Ohori and Peter Buneman. Static type inference for parametric classes. In *Proc. of ACM OOPSLA*, pages 445–456, Oct. 1989.

[OBBT89] Atsushi Ohori, Peter Buneman, and Val Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proc. of ACM SIGMOD*, pages 46–57, May.–Jun. 1989.

[Oho92] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *Proc. of ACM POPL*, pages 154–165, Jan. 1992.

[OT94] Atsushi Ohori and Keishi Tajima. A polymorphic calculus for views and object sharing. In *Proc. of ACM PODS*, pages 255–266, May 1994.

[Qia94] Xiaolei Qian. Inference channel-free integirity constraints in multilevel relational databases. In *Proc. of IEEE Symp. on Research in Security and Privacy*, pages 158–167, 1994.

[QSK+93] Xaiolei Qian, Mark E. Stickel, Peter D. Karp, Teresa F. Lunt, and Thomas D. Garvey. Detection and elimination of inference channels in multilevel relational database systems. In *Proc. of IEEE Symp. on Research in Security and Privacy*, pages 196–205, 1993.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. of ACM POPL*, pages 77–88, Jan. 1989.

[RBKW91] Fausto Rabitti, Elisa Bertino, Won Kim, and Darrell Woelk. A model of authorization for next-generation database systems. *ACM TODS*, 16(1):88–131, Mar. 1991.

[Row89] Neil C. Rowe. Inference-security analysis using resolution theorem-proving. In *Proc. of IEEE ICDE*, pages 410–416, Feb. 1989.

[RS91] Joel Richardson and Peter Schwarz. Aspects: Extending objects to support multiple, independent roles. In *Proc. of ACM SIGMOD*, pages 298–307, May 1991.

[Run92] Elke A. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proc. of VLDB*, pages 187–198, Aug. 1992.

[SLT91] Marc H. Scholl, Christian Laasch, and Markus Tresch. Updatable views in object-oriented databases. In *Proc. of Int. Conf. on Deductive and Object-Oriented Database Systems*, volume 566 of *LNCS*, pages 189–207. Springer-Verlag, Dec. 1991.

[SÖ91] Tzong-An Su and Gultekin Özsoyoglu. Controlling FD and MVD inferences in multilevel relational database systems. *IEEE Trans. on Know. and Data.*, 3(4):474–485, Dec. 1991.

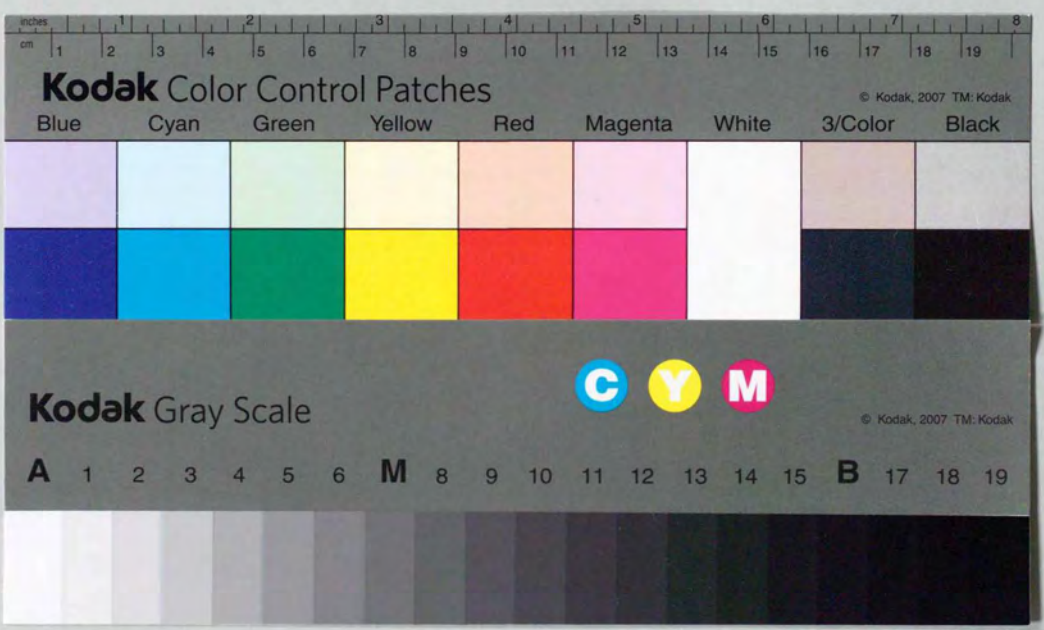
[Spo89] David L. Spooner. The impact of inheritance on security in object-oriented database systems. In *Database Security II: Status and Prospectus*, pages 141–150. Elsevier Science Publ., 1989.

[Taj96] Keishi Tajima. Static detection of security flaws in object-oriented databases. In *Proc. of ACM SIGMOD*, pages 341–352, Jun. 1996. to appear.

[TYI88] Katsumi Tanaka, Masatoshi Yoshikawa, and Kozo Ishihara. Schema virtualiztion in object-oriented databases. In *Proc. of IEEE ICDE*, pages 23–30, Feb. 1988.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proc. of IEEE LICS*, pages 37–44, Jun. 1987.

[Wie86] Gio Wiederhold. Views, objects, and databases. *IEEE Computer*, 19(12):37–44, Dec. 1986.


inches
cm
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19