Efficient Dynamic Look-Up Strategy for Multi-Methods

Weimin Chen*, Volker Turau**, and Wolfgang Klas*

*GMD-IPSI
Integrated Publication and Information Systems Institute
Dolivostr. 15
64293 Darmstadt, Germany
Email: {chen, klas}@darmstadt.gmd.de

**Fachhochschule Giessen-Friedberg
Fachbereich MND
Wilhelm-Leuschner-Str. 13
61169 Friedberg, Germany
Email: turau@courbet.fh-friedberg.de

Abstract. In object-oriented programming languages, multiple-dispatching provides increased expressive power over single-dispatching by guiding method look-up using the values of all arguments instead of the receiver only. There have been several programming languages supporting this mechanism and they demonstrate its usefulness. However, efficient implementation of multi-methods is still critical with regard to its success as a standard. In this paper, we present a new mechanism for implementing multi-methods dynamic lookup based on automaton techniques. Analysis and experimental results show that our strategy is time and space efficient. The presented result can provide the basis for designing new object-oriented paradigms based on multi-methods.

1 Introduction

Today most programming languages are based on the notion of types. A data type consists of a representation and a set of operations which can be applied to instances of the types. In many object-oriented languages, types are organized in a hierarchy and a subtype relation is defined over them. One important feature of this subtype relation is the *subtype polymorphism* [7]: if A is a subtype of B, then every instance of A is also an instance of B. Operations on the instances of types are defined by *generic functions*, where a generic function corresponds to a set of *methods* and the methods define the type-specific behavior of the generic function. In the presence of subtype polymorphism, method selection must occur at run time.

In many object-oriented languages, a message (function invocation) is sent to a distinguished receiver object, and the run time "type" of the receiver determines the method that is actually invoked by the message. The arguments of the message are passed on to the invoked method but do not participate in the method dispatching. For example, in C++ we can define a virtual function of the form `float`

`area(shape)`[1] which is dynamically dispatched based on the actual type of `shape` supplied with the function invocation. However, one cannot write a virtual function of the form `displayOn(shape,device)` which is dynamically dispatched based on actual types of both `shape` and `device`.

To surmount these limitations, some object-oriented languages include a more powerful form of function invocation in which all arguments of a method can participate in the method dispatching (method lookup), i.e. a method is dynamically dispatched based on the types of all arguments. These methods are called *multi-methods*. The dispatching for multi-methods is called *multiple-dispatching*[8]. Perhaps the most-known languages that support multi-methods are CLOS [6] and one of its predecessors CommonLoops [5].

One fundamental issue for multi-methods is the efficient mechanism for method lookup. Efficient implementation of multi-methods is still critical with regard to its success as a standard. There have been several time-efficient lookup mechanisms proposed [11, 13, 16]. However, by far the largest problem is that all these structures for dynamic lookup may lead to combinatorial explosion, which leads to a space problem. Based on the result of Agrawal, *et al.* [2] on static type checking of multi-methods, in this paper we present a new mechanism for multi-method lookup using lookup automatons. If n is the arity of the function invocation, then the time-complexity of the method lookup is $O(n)$. The main contribution of our approach compared to the other lookup mechanisms is that it is more space-efficient while providing the same time-efficiency as [11, 13, 16]. The results presented are intended to provide a general technique to optimize the performance of dispatching for multi-methods.

The organization of this paper is as follows. We state the basic concepts concerning the type hierarchy and multi-methods in section 2. Section 3 describes the overall approach in order to have an intuitive idea of our approach. The formal statement of the problem is presented in section 4. Next, sections 5 and 6 discuss algorithms to construct and simulate the lookup automaton respectively. Experimental results are discussed in section 7. We discuss the related work in section 8. Finally section 9 summarizes the results of our work.

2 Basic Concepts

2.1 Type Hierarchy and Type Ordering

In the discussion that follows, we represent the subtype relation by $\preccurlyeq$ in a type hierarchy $\mathcal{T}$. Also we denote $A \prec B$ if $A \preccurlyeq B$ and $A \neq B$, in this case we say that A is *subtype* of B, or B is *supertype* of A. Particularly, we use $A \vec{\prec} B$ to denote that B is the *direct supertype* of A. Since relation $\prec$ defines a partial order, such a system of types forms a directed acyclic graph (DAG). There is a path from A to B if and only if $A \prec B$. In the rest of this paper, a type A in $\mathcal{T}$ is denoted by $A \in \mathcal{T}$, and S, a subset of types in $\mathcal{T}$, is denoted by $S \subseteq \mathcal{T}$.

A *local type ordering for a type* C is a total order $\propto_C$ over C and its supertypes such that if $C \prec A$ or $C \prec B$, then $C \propto_C A$, $C \propto_C B$, and either $A \propto_C B$ or $B \propto_C A$. Fur-

1. the first argument can be understood as the method receiver.

thermore, if $C \prec B$, and if $D \propto_B E$ in the local type ordering for type B, then $D \propto_C E$ in the local type ordering for the type C, and this rule is recursively applied. This rule must be based on the restriction that there must not exist another type $B' \neq B$ such that $C \prec B'$ and $E \propto_{B'} D$. If this restriction is satisfied, $\mathcal{T}$ is called *consistent*. In this paper, we always assume that $\mathcal{T}$ is consistent.

CLOS [6] is an example of a language that uses local type ordering to determine the type precedence. In this paper, we discuss the general lookup techniques for languages which use the local type ordering to determine the type precedence, while noticing possible simplified cases if $\mathcal{T}$ is a directed tree (single inheritance).

2.2 Method Applicability, Confusability, Specificity, and Consistency

The main concepts concerning multi-methods which we use in this paper are presented in [2]:

- *method applicability* — given a generic function invocation, $m(T_1,\ldots,T_n)$, we say that a method $m_k(T_1^k,\ldots,T_n^k)$ is *applicable* for that invocation if and only if $T_i \preccurlyeq T_i^k$ for $1 \leq i \leq n$;
- *method confusability* — if two methods are both applicable for a function invocation, we say that they are *confusable*. Formally, methods $m_1(T_1^1,\ldots,T_n^1)$ and $m_2(T_1^2,\ldots,T_n^2)$[2] are *confusable* if $\forall\, i,\ 1 \leq i \leq n$, there exists a type T_i, such that $T_i \preccurlyeq T_i^1$ and $T_i \preccurlyeq T_i^2$; otherwise they are *non-confusable*. The equivalence classes of the transitive extension of the relation *confusable* are called *confusable sets*. If $\mathcal{M}\flat$ is a *confusable set* of methods and m_i and m_j are in $\mathcal{M}\flat$ then there are k (≥ 0) methods $m_1, m_2, \ldots, m_k$ in $\mathcal{M}\flat$ such that m_i is confusable with m_1, m_1 is confusable with m_2, ..., and m_k is confusable with m_j. We say that a generic function invocation $m(T_1,\ldots,T_n)$ is *covered by* a confusable set $\mathcal{M}\flat$ if there exists a method $m' \in \mathcal{M}\flat$ such that m' is applicable for m;
- *method specificity* — if one method has precedence over another for a given invocation, we say that it is *more specific* than the other. One mechanism is called *inheritance order precedence*: while suppose $m_i(T_1^i,\ldots,T_n^i)$ and $m_j(T_1^j,\ldots,T_n^j)$ are two applicable methods for a generic function invocation $m(T_1,\ldots,T_n)$, we consider their formal arguments in a prespecified order (such as left-to-right[3]), and find the first argument position in which the formal argument types of m_i and m_j differ, say k. If $T_k^i \propto_{T_k} T_k^j$, then m_i is more specific than m_j, and vice versa. The inheritance order precedence is sufficient to determine method specificity in a multiple inheritance language with multi-methods [2]. CLOS is an example of a language that uses inheritance order precedence to determine method specificity.

---

2. For the sake of easy description, the method names are subscripted. In real system, however, the confusable methods' names are identical.

3. Any prespecified order can be transformed as left-to-right by exchanging the argument location during compile time. Hence, without loss of generality, we can assume that the prespecified order is left-to-right during dispatching.

- *method consistency* — two methods $m_i(T_1^i,\ldots,T_n^i) \rightarrow R_i$ (R_i denotes the type of the result) and $m_j(T_1^j,\ldots,T_n^j) \rightarrow R_j$ of a generic function M are *mutually consistent* if whenever they are both applicable for arguments of types $T_1,\ldots,T_n$ and m_i is more specific than m_j, then $R_i \preccurlyeq R_j$. A generic function is *consistent* if all its methods are mutually consistent.

2.3 An Important Result

An important result presented in [2] for static type checking of multi-methods is the following.

THEOREM 1. A generic function invocation $m(T_1,\ldots,T_n)$ is covered by at most one confusable set, which can be determined at compile time. ■

Based upon this result, the task of dynamic dispatching is to find the most specific method (in the confusable set) for the given generic function invocation. In fact, for the purpose of dynamic dispatching, only the following property concerning confusable sets is used: all methods in a confusable set have the same name as well as the same arity. Although in this paper we discuss the dispatching approach in the context of statically typed languages, yet it is still valid for dynamically typed languages where the concept of "confusable set" can be understood as "the set of all methods with the same name as well as the same arity".

3 Overview of Our Dispatching Approach

In order to have an intuitive idea of our approach, in this section we view the overall approach of multiple-dispatching by presenting examples. We introduce a *lookup automaton* (*LUA*) to simulate the dynamic dispatching in a given confusable set. An LUA is a deterministic finite automaton [3] and is defined as 5-tuple $\mathfrak{D} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of *states*; Σ is a finite set of *input symbols*; δ is a *state transition function*, which is a mapping $Q \times \Sigma \rightarrow Q$; $q_0 \in Q$ is the *initial state* of the finite state control; and $F \subseteq Q$ is the set of *final states*.

Fig. 1 shows an example of a type hierarchy $\mathcal{T}$, a confusable set $\mathcal{M}_6$, and the corresponding LUA. In the LUA, we indicate the method precedence order at each state by *a list of sets of methods*. Suppose that at state q the list is $a_1a_2 \ldots a_l$ (where $a_i \subseteq \mathcal{M}_6$), and method $m_1 \in a_i$ and $m_2 \in a_j$. If $i < j$, then method m_1 at state q will have higher priority to be selected than m_2; if $i = j$, then m_1 and m_2 at state q will have the same priority to be selected. For example, consider the list $\{m_2, m_3\}\{m_1\}$ at state q_2: methods m_2 and m_3 have higher priority to be selected than the other method m_1, while m_2 and m_3 have the same priority to be selected.

The basic idea to construct the LUA is the following: in order to reduce the size of the LUA, the number of states introduced should be as small as possible. Informally, if states have the same precedence order, they will be "merged" together. In order to easily catch the idea, we firstly state how to simulate this LUA. Afterwards, we describe the overall approach of how to construct it.

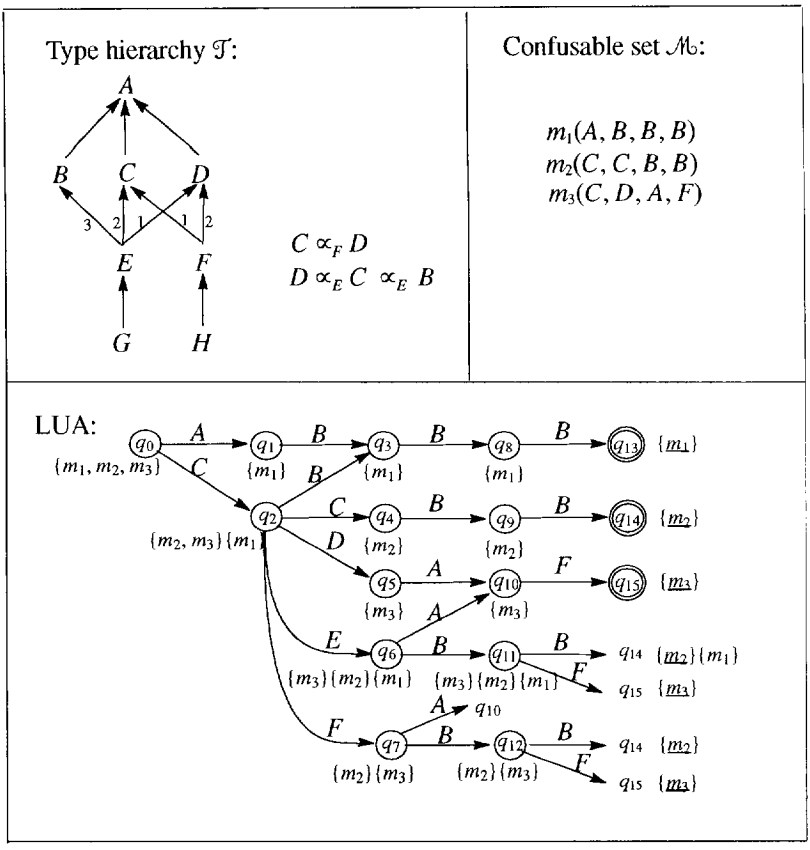

Fig. 1. An example of a type hierarchy, a confusable set, and the corresponding LUA

In the discussion of this paragraph, we use upper case letters to denote types and the corresponding lower case letters to denote their instances. For example, we write a to denote an instance of type A. Now we consider a generic function invocation $m(e, h, c, f)$. Initially we are at start state q_0. There are two types A and C following state q_0. We select transition $\delta(q_0, C) = q_2$ because, in the set $\{A, C\}$, C is *the most specific supertype* of the first argument type E. At state q_2, five types B, C, D, E, and F follow. We select transition $\delta(q_2, F) = q_7$ because, in the set $\{B, C, D, E, F\}$, F is *the most specific supertype* of the second argument type H. Similarly, the next state transitions are $\delta(q_7, A) = q_{10}$ and $\delta(q_{10}, F) = q_{15}$. At the final state q_{15}, method m_3 is the most specific method for the given invocation. Note that, during the LUA simulation, only the knowledge of relation $\preccurlyeq$ is needed. The number of state transitions is equal to the function arity 4.

The LUA in Fig. 1 is constructed as follows. At first, the initial state q_0 is created, where three methods m_1, m_2, and m_3 may be applicable with the same priority, i.e. at q_0 the method precedence order is $\{m_1, m_2, m_3\}$. At the next level, the set of the first argu-

ment types of the methods m_1, m_2, and m_3 is $\{A, C\}$. Consider the state transitions $\delta(q_0, A)$ and $\delta(q_0, C)$. At the state $\delta(q_0, A)$, only method m_1 may be applicable, so that we create a new state $q_1 = \delta(q_0, A)$ with precedence order $\{m_1\}$; at state $\delta(q_0, C)$, three methods m_1, m_2, and m_3 may be applicable but m_2 and m_3 have higher priority to be selected than m_1, i.e. the precedence order is $\{m_2, m_3\}\{m_1\}$. At this level, so far there is still no state with precedence order $\{m_2, m_3\}\{m_1\}$. Thus, a new state $q_2 = \delta(q_0, C)$ with precedence order $\{m_2, m_3\}\{m_1\}$ is created. Now consider the situation at the next level. At state q_1, only m_1 may be applicable, so that only state transition $\delta(q_1, B)$ is considered (B is the second argument type of m_1). We create a new state $q_3 = \delta(q_1, B)$ with precedence order $\{m_1\}$. At state q_2, the possible applicable methods are m_1, m_2, and m_3, so the state transitions $\delta(q_2, B)$, $\delta(q_2, C)$, and $\delta(q_2, D)$ must be considered. Since the state $\delta(q_2, B)$ is with precedence order $\{m_1\}$, which is the same as the precedence order at state q_3, we define $\delta(q_2, B) = q_3$. Similar to the situation of state transition $\delta(q_0, C)$, the new states $q_4 = \delta(q_2, C)$ and $q_5 = \delta(q_2, D)$ are created. Now, at q_2 we have introduced the transitions $\delta(q_2, B)$, $\delta(q_2, C)$, and $\delta(q_2, D)$. In order to simulate all possible cases, however, it is still necessary to introduce additional transitions $\delta(q_2, E)$ and $\delta(q_2, F)$. We will explain this in the next paragraph. Going on with this procedure, at last we construct the whole LUA, where the final states are with double circles, and the most specific methods at final states are underscored.

Now we turn back to discuss why at state q_2 it is necessary to introduce the additional transitions $\delta(q_2, E)$ and $\delta(q_2, F)$. Consider the case that these additional transitions are not introduced. Recall that m_3 is the most specific method for the above invocation $m(e, h, c, f)$. For this function invocation we consider the LUA simulation again. At first, for the first argument type E, we select transition $\delta(q_0, C) = q_2$. Consider the second argument type H. As types E and F are not introduced at q_2, in the set of the remaining types $\{B, C, D\}$, finding the most specific supertype of H needs knowledge of the relation $\propto_H$ rather than the relation $\preccurlyeq$. By the relationship $C \propto_H D$ (since $C \propto_F D$ holds and $\mathcal{T}$ is consistent), we know that, in set $\{B, C, D\}$, C is the most specific supertype of H with respect to $\propto_H$. Thus, at state q_2 the state transition $\delta(q_2, C) = q_4$ is selected. Clearly the simulation is not correct — since eventually the most specific method m_3 is not in the path following q_4. Therefore, a backtracking is required to re-select the state transition as $\delta(q_2, D)$ instead of $\delta(q_2, C)$. Hence, if the types E and F are not introduced at state q_2, there will be two consequences during the simulation:

- the knowledge of relation $\propto_H$ instead of relation $\preccurlyeq$ is needed, and
- the backtracking is required.

Given a confusable set and a type hierarchy, there is another possibility to construct the LUA. At first, one can simply enumerate all possible dispatching cases and then represent that by a finite automaton. Afterwards, standard techniques to minimize this automaton can be employed. This seems to be a natural way for constructing the LUA. However, there are some problems with this approach when used for larger systems. The main weakness is that, before minimizing the automaton, a large memory overhead may take place because representing all possible dispatching cases may lead

the automaton to be combinatorial explosive.[4] Consequently, the procedure to minimize the automaton can become very expensive. In an approach presented in this paper we overcome this weakness by addressing the construction and minimization of the LUA simultaneously: the procedure to reduce the size of the LUA is performed at the time of constructing the LUA. Consequently, with our approach the large memory overhead is prevented because unnecessary states of the LUA are never created.

4 Formal Statement of the Problem

In this section, we present the terminology and notation which we will be employing throughout the paper. Moreover, we indicate the several associated results.

4.1 The Operator $\sqcap$ and the Function $M^*_{(k)}$

In the following we always assume that $\mathcal{M}$ is a confusable set of methods with arity n. Let $\mathfrak{R}$ be the set $\{ a_1a_2...a_p \mid p>0, a_i \subseteq \mathcal{M}, a_i \cap a_j = \varnothing, \text{for } i \neq j\}$. The list $a_1a_2...a_p$ is represented by $\prod_{i=1}^{p} a_i$ and each a_i $(i = 1, ..., n)$ is called a factor of $\prod_{i=1}^{p} a_i$. Note that the order among the factors is important, e.g. $a_1a_2 \neq a_2a_1$. An operator $\sqcap: \mathfrak{R} \times \mathfrak{R} \rightarrow \mathfrak{R}$ is defined by $A \sqcap B = \prod_{i=1}^{q}\prod_{j=1}^{p}(a_i \cap b_j)$, for $A = \prod_{i=1}^{p} a_i$ and $B = \prod_{i=1}^{q} b_i \in \mathfrak{R}$. Clearly $A \sqcap B \in \mathfrak{R}$. In this paper, we always assume that all factors of elements in $\mathfrak{R}$ are not empty. Therefore, if a factor in $A \sqcap B$ is empty we omit it. For $A, B \in \mathfrak{R}$, we say $A = B$ if and only if they have the same factor list after omitting all empty factors. A function *domain*: $\mathfrak{R} \rightarrow 2^{\mathcal{M}}$ is defined by $domain(A) = \bigcup_{i=1}^{p} a_i$, for $A = \prod_{i=1}^{p} a_i \in \mathfrak{R}$. Let m_1, $m_2 \in domain(A)$ then we say that $m_1 \in a_i$ precedes $m_2 \in a_j$ with respect to A, if $i < j$.

The function $T_{(k)}$: $\mathcal{M} \rightarrow \mathcal{T}$ is defined to map a method to its k^{th} argument type, i.e. $T_{(k)}(m(t_1, t_2, \ldots, t_n)) = t_k$, for $m(t_1, t_2, \ldots, t_n) \in \mathcal{M}$. An inverse function $T^{-1}_{(k)} : \mathcal{T} \rightarrow 2^{\mathcal{M}}$ can be defined by means of $T^{-1}_{(k)}(t) = \{m \in \mathcal{M} \mid T_{(k)}(m) = t\}$. Moreover, we define a function $M^*_{(k)}$: $\mathcal{T} \times 2^{\mathcal{T}} \rightarrow \mathfrak{R}$ by means of $M^*_{(k)}(t, S) = \prod_{i=1}^{l} T^{-1}_{(k)}(t_i)$, where $(t_1, \ldots, t_l)$ is a type precedence list over t with respect to the set $\{t' \in S \mid t \preccurlyeq t'\}$ such that $t_i \propto_t t_{i+1}$, for $1 \leq i < l$.

PROPOSITION 2. $\forall A, B$, and $C \in \mathfrak{R}$,, the following holds:

(a) (*associativity*) $(A \sqcap B) \sqcap C = A \sqcap (B \sqcap C)$;
(b) (*conditional commutativity*) $A \sqcap B = B \sqcap A$, if A or B consists of only one factor;
(c) (*identity element*) $A \sqcap \mathcal{M} = \mathcal{M} \sqcap A = A$, if we regard $\mathcal{M} \in \mathfrak{R}$;

4. In section 7, experimental results will show the degree of this combinatorial explosion.

(d) $domain(A \sqcap B) = domain(A) \cap domain(B)$;

(e) $\forall\, t \in \mathcal{T}$, $M^*_{(k)}(t, T_{(k)}(\mathcal{M}_b)) \in \mathcal{R}_b$ represents a precedence order for the methods (in $\mathcal{M}_b$) whose k^{th} argument types are the supertypes of t;

(f) $\forall\, t \in \mathcal{T}$ and $\forall\, S \subseteq \mathcal{T}, A \sqcap M^*_{(k)}(t, S) = A \sqcap M^*_{(k)}\big(t, S \cap T_{(k)}(domain(A))\big)$.

PROOF. (a), (b), (c), (d), and (e) can be directly derived from the corresponding definitions.

(f) $\forall\, s \in S - T_{(k)}(domain(A))$, we have $domain(A) \cap T^{-1}_{(k)}(s) = \emptyset$. By definition, $M^*_{(k)}(t, S) = \prod_{i=1}^{l} T^{-1}_{(k)}(t_i)$, where $(t_1, \ldots, t_l)$ is a type precedence list over t with respect to the set $\{t' \in S \mid t \leqslant t'\}$ such that $t_i \propto_t t_{i+1}$, for $1 \leq i < l$. Suppose that $\{t_1, \ldots, t_l\} \cap T_{(k)}(domain(A)) = \{t_{i_1}, \ldots, t_{i_q}\}$ and $A = \prod_{i=1}^{p} a_i$. Thus, $\forall\, t' \in \{t_1, \ldots, t_l\} - \{t_{i_1}, \ldots, t_{i_q}\}$, we have $domain(A) \cap T^{-1}_{(k)}(t') = \emptyset$. Therefore, $A \sqcap M^*_{(k)}(t, S) = \prod_{i=1}^{p} \prod_{j=1}^{l} \big(a_i \cap T^{-1}_{(k)}(t_j)\big) = \prod_{i=1}^{p} \prod_{j=1}^{q} \Big(a_i \cap T^{-1}_{(k)}(t_{i_j})\Big) = A \sqcap M^*_{(k)}\big(t, S \cap T_{(k)}(domain(A))\big)$. ■

The operator $\sqcap$ and the functions *domain* and $T_{(k)}$ are easy to implement. However, the implementation of $M^*_{(k)}(t, S)$ will rely on the algorithm of how to determine the type precedence list $(t_1, \ldots, t_l)$ over t with respect to the set $\{t' \in S \mid t \leqslant t'\}$ such that $t_i \propto_t t_{i+1}$ for $1 \leq i < l$. Ducournau, *et al.* [10] discussed this supertype linearization problem in general. In the context of CLOS, however, Bobrow, *et al.* [6] proposed a concrete algorithm to determine the type precedence list. For every $c \in \mathcal{T}$, define $R_c = \{(c, c_1), (c_1, c_2), \ldots, (c_{k-1}, c_k)\}$, where $c_1, \ldots, c_k$ are the direct supertypes of c, and $c_i \propto_c c_{i+1}$ for $1 \leq i < k$. Let T_c be the set of type c and its supertypes. Let $R = \bigcup_{c \in T_t} R_c$. To compute the precedence list for t, topological sorting proceeds by finding a type c in T_t such that no other type precedes that element according to the elements in R. The type c is placed first in the result. Remove c from T_t, and remove all pairs of the form (c, c'), $c' \in T_t$, from R. Repeat the process, adding types with no predecessors to the end of the result. Stop when no element can be found that has no predecessor. Using this approach, we compute the type precedence list $(t_1, \ldots, t_l)$ over t with respect to T_t. Therefore, in order to compute $M^*_{(k)}(t, S)$, we can extract all $t_i \in S$ from the list $(t_1, \ldots, t_l)$. If $\mathcal{T}$ is a directed tree, the algorithm will be much simpler. We will not describe these algorithms any further.

4.2 The Functions *GLB*, *closure*, and $LUB_\diamond$

Given two types $s, t \in \mathcal{T}$, we define the set of *greatest lower bounds* for s and t by

$GLB(s, t) = \{u \in \mathcal{T} \mid u \preccurlyeq s, u \preccurlyeq t$, and $\neg\exists\, u' \in \mathcal{T}$ such that $u \prec u' \preccurlyeq s$ and $u \prec u' \preccurlyeq t\}$.[5]

For a subset of types $S \subseteq \mathcal{T}$, we say S *closed* if $\forall\, s, t \in S$, $GLB(s, t) \subseteq S$. The *closure* of a subset S, denoted by *closure*(S), is defined as the intersection of all closed subsets T such that $S \subseteq T \subseteq \mathcal{T}$. Clearly *closure*($S$) is also closed.

On the other hand, $\forall\, t \in \mathcal{T}$ and $S \subseteq \mathcal{T}$, we define the set of *least upper bounds* for t and S by

$LUB_{\diamond}(t, S) = \{s \in S \mid t \preccurlyeq s, \neg\exists\, s' \in S$ such that $t \preccurlyeq s' \prec s\}$.[6]

For example, in the type hierarchy $\mathcal{T}$ shown in Fig. 1, $GLB(C, D) = \{E, F\}$, *closure*($\{C, D\}$) = $\{C, D, E, F\}$, and $LUB_{\diamond}(G, \{A, C, D\}) = \{C, D\}$.

The following propositions are the direct results of the definitions of *GLB* and $LUB_{\diamond}$.

PROPOSITION 3. Let $s, t_1, t_2 \in \mathcal{T}$ such that $s \preccurlyeq t_1$ and $s \preccurlyeq t_2$. Then $\exists\, s' \in GLB(t_1, t_2)$ such that $s \preccurlyeq s'$. ■

PROPOSITION 4. Let $t \in \mathcal{T}$, $S \subseteq \mathcal{T}$, and $s \in S$. If $t \preccurlyeq s$, then $\exists\, s' \in LUB_{\diamond}(t, S)$ such that $t \preccurlyeq s' \preccurlyeq s$. ■

THEOREM 5. Let $S \subseteq \mathcal{T}$. Then S is closed if and only if $\forall\, t \in \mathcal{T}$, $LUB_{\diamond}(t, S)$ contains at most one element.

PROOF. *if*: Let $s_1, s_2 \in S$. It suffices to prove that $GLB(s_1, s_2) \subseteq S$. Let $s \in GLB(s_1, s_2)$. By proposition 4, $\exists\, s_1', s_2' \in LUB_{\diamond}(s, S)$ such that $s \preccurlyeq s_i' \preccurlyeq s_i$ for $i = 1, 2$. By assumption, $LUB_{\diamond}(s, S)$ contains at most one element. Hence, $s_1' = s_2'$. Now $s \in GLB(s_1, s_2)$ implies that $s = s_1' \in LUB_{\diamond}(s, S) \subseteq S$.

only if: Let S be closed and $t \in \mathcal{T}$. Furthermore, let $s_1, s_2 \in LUB_{\diamond}(t, S)$. Clearly if $t \in S$ then $s_1 = s_2 = t$. Thus, we assume that $t \notin S$. This induces $t \notin GLB(s_1, s_2)$, as S is closed. Furthermore, we have $t \preccurlyeq s_1$ and $t \preccurlyeq s_2$. Therefore, by proposition 3, $\exists\, s \in GLB(s_1, s_2)$ such that $t \prec s$. Hence, by the definition of $LUB_{\diamond}$, $s = s_1$ and $s = s_2$ hold. This implies that $\| LUB_{\diamond}(t, S) \| \leq 1$. ■

The following is the algorithm for *closure* and the algorithm for the function $LUB_{\diamond}$. Initially, a topological sorting is performed on the types of $\mathcal{T}$ with respect to the relation $\prec$. Let τ be the number of types of $\mathcal{T}$ and let $T[1]$, $T[2]$, ..., and $T[\tau]$ be the types of $\mathcal{T}$ such that $T[i] \prec T[j]$ holds only if $i < j$.

In the function *closure*, a variable μ of type array[$1 .. \tau$] of {*not_in_closure*, *in_closure*} is needed. Initially, $\mu[i]$ = *not_in_closure*, for $1 \leq i \leq \tau$.

---

5. In order to keep things simple, we borrow the notation *GLB* which is originally defined in *lattice* structures, where *GLB* is a unique element. However, in a DAG, we define *GLB* to be a set.

6. In order to distinguish our notion from the well-known standard notion *LUB* defined in lattice structure, we use the notation $LUB_{\diamond}$.

```
function closure(S)
1  {  for i = τ downto 1 do
2        if T[i] ∈ S then μ[i] ← in_closure
3        else
4        {  Δ ← {j | T[i] ≺→ T[j], and μ[j] = in_closure}
5           if Δ ≠ ∅ then
6           {  j_min ← min Δ
7              if ∃ j ∈Δ − {j_min}, such that ¬(T[j_min] ≺ T[j]) then
8                 μ[i] ← in_closure
9              else  /* ∀ j ∈Δ − {j_min}, T[j_min] ≺ T[j]. So T[i] ∉ closure(S).
                        𝒯 is updated in the following. */
10                foreach T[i'] ∈𝒯 such that T[i'] ≺→ T[i] do
11                   delete the edge from T[i'] to T[i] and add the edge from
                     T[i'] to T[j_min]
12          }
13       }
14    return {T[j] | μ[j] = in_closure, 1 ≤ j ≤τ}
15 }
```

The function uses a *for-loop* (lines 1-13) to scan all types of $\mathcal{T}$, where $\mathcal{T}$ may be updated (c.f. lines 10-11). Let $\mathcal{T}_k$ ($1 \le k \le \tau$) be the updated $\mathcal{T}$ when the for-loop has been executed k times. In $\mathcal{T}_k$, we denote the relations $\prec$, $\preccurlyeq$, and $\vec{\prec}$ by $\prec_k$, $\preccurlyeq_k$, and $\vec{\prec}_k$. Therefore, in the function *closure*, the relations $\prec$ and $\vec{\prec}$ (used at lines 4, 7, and 10) must be understood as $\prec_{i-1}$ and $\vec{\prec}_{i-1}$ in the corresponding $\mathcal{T}_{i-1}$. Now we define

$$\tilde{T}_k = \{T[i] \in \mathcal{T}_k \mid \mu[i] = in\_closure \text{ or } i < k\}.$$

Before proving the correctness of the function *closure*, it is necessary to introduce the following lemma.

LEMMA 6. When the for-loop of function *closure* has been executed k times, the following properties hold:

(a) $\forall\, a, b \in \tilde{T}_k$, the relation $a \prec_k b$ holds if and only if the relation $a \prec b$ in $\mathcal{T}$ holds;

(b) Given a number k, $1 < k \le \tau$, let $T[j] \in \tilde{T}_k$ with $\mu[j] = in\_closure$ and $T[k-1] \prec_k T[j]$. There must be a number j', $k \le j' \le j$, such that $T[k-1] \vec{\prec}_k T[j'] \preccurlyeq_k T[j]$ and $\mu[j'] = in\_closure$. ■

THEOREM 7. The function *closure* returns the correct result. ■

The proofs of lemma 6 and theorem 7 are stated in the appendix.

Remarks:

(a) If $\mathcal{T}$ is a directed tree (single inheritance), then every subset $S \subseteq \mathcal{T}$ is always closed, i.e. $closure(S) = S$ holds;

(b) As we have mentioned, at line 7 the relation $\prec$ must be understood as $\prec_{i-1}$ in $\mathcal{T}_{i-1}$. By the definition of Δ (c.f. line 4), we have $T[j] \in \tilde{T}_{i-1}$, for $j \in \Delta$. According to lemma 6(a), at line 7 the relation $\prec_{i-1}$ can be treated as the relation $\prec$ in

$\mathcal{T}$. In order to make subtype relationships tests fast, Agrawal, *et al.* [1] proposed a structure to maintain a compressed transitive closure of the subtype relation. An index and a set of ranges are associated with each type. If the index of one type falls into a range of another type, then the first type is a subtype of the second. Using this technique, we can test subtype relationships in constant time if the type hierarchy is a directed tree (single inheritance). If the type hierarchy is a DAG (multiple inheritance), experimental results show that subtype relations can be tested in essentially constant time [1], and, in the worst case, in $O(log(\tau))$ time, where τ is the number of types in the type hierarchy;

(c) As the type hierarchy may be updated, it is necessary to make a copy at the beginning. Furthermore, at line 10, it is necessary to find all direct subtypes for a given type. Therefore, the data structure for the copy of the type hierarchy must efficiently support this operation;

(d) A topological sort of the types of $\mathcal{T}$ is performed only once. This sorting will be used in later stages;

(e) The time-complexity of the function *closure* is $O(\tau + e)$, where e is the number of edges in $\mathcal{T}$.

In later sections, we will see that the function $LUB_{\diamond}(s, S)$ is invoked at run time, only in the case that S is closed. The following is the algorithm of $LUB_{\diamond}(s, S)$ under the condition that S is closed.

```
function LUB◇(s, S)
1  {  Suppose S = {T[i1], T[i2], ... , T[ik]}, where 1 ≤ i1 < i2 < ... < ik ≤τ
2     Suppose s = T[l]
3     for j = 1 to k do
4        if ij ≥ l and s≤T[ij] then return {T[ij]}
5     return ∅
6  }
```

Remarks:

(a) The correctness of the function $LUB_{\diamond}$ is a direct result of theorem 5, i.e. the result of $LUB_{\diamond}(s, S)$ contains at most one element;

(b) The worst time-complexity of the function $LUB_{\diamond}$ is $O(\|S\|)$.

4.3 The Central Result

Having introduced the basic concepts, we can now state the central result.

THEOREM 8. Given the types $t_1, \ldots, t_k \in \mathcal{T}$, let

$$R_k = R(t_1, \ldots, t_k) = \prod_{i=1}^{k} M^*_{(i)}\left(t_i, T_{(i)}(\mathcal{M}_b)\right). \tag{1}$$

(a) Let a be a factor of R_k and $m \in a$. Then $a = \{m' \in \mathcal{M}_b \mid m \text{ and } m' \text{ have the same first } k \text{ argument types}\}$.

(b) $R(t_1, \ldots, t_k)$ represents the precedence order of the methods (in $\mathcal{M}_b$) whose first k argument types are respectively the supertypes of $t_1, \ldots, t_k$.

PROOF. (a) Let $m_1(t_1^1,\ldots,t_1^k,\ldots)$ and $m_2(t_2^1,\ldots,t_2^k,\ldots) \in \mathfrak{R}$. Suppose that $\exists\, i,\ 1 \le i \le k$ such that $t_1^i \ne t_2^i$. Hence, methods m_1 and m_2 are not together in a factor of $M^*_{(i)}\left(t_i, T_{(i)}(\mathcal{M})\right)$ nor in a factor of R_k. On the other hand, it is clear that if m and m' have the same first k argument types, then they are both together in one factor.

(b) The proof is by induction on k. When $k = 1$, then $R(t_1) = M^*_{(1)}\left(t_1, T_{(1)}(\mathcal{M})\right)$ and the result follows from proposition 2(e). By definition, we have $R(t_1,\ldots,t_k) = R(t_1,\ldots,t_{k-1}) \sqcap M^*_{(k)}\left(t_k, T_{(k)}(\mathcal{M})\right)$. Suppose $R_{k-1} = R(t_1,\ldots,t_{k-1})$ defines the precedence order of the methods (in $\mathcal{M}$) whose first k–1 argument types are respectively the supertypes of $t_1,\ldots,t_{k-1}$. By proposition 2(e), $M^*_{(k)}\left(t_k, T_{(k)}(\mathcal{M})\right)$ (abbreviated by $M^*_{(k)}$) presents the precedence order of the methods (in $\mathcal{M}$) whose k^{th} argument types are the supertypes of t_k. Let $X_k \in \mathfrak{R}$ be the precedence order of the methods (in $\mathcal{M}$) whose first k argument types are respectively the supertypes of $t_1,\ldots,t_k$. Note that X_k satisfies the following properties:

- $domain(X_k) = domain(R_{k-1}) \cap domain(M^*_{(k)})$;
- let $m_1, m_2 \in domain(R_{k-1}) \cap domain(M^*_{(k)})$. If m_1 and m_2 are in different factors of R_{k-1}, then m_1 and m_2 will be in different factors of X_k and the order between them must be kept. On the other hand, if m_1 and m_2 are in a same factor of R_{k-1}, then m_1 and m_2 have the same first k–1 argument types. Therefore, the order between them presented in X_k will be determined by their order presented in $M^*_{(k)}$.

Hence, by means of the operator $\sqcap$, $X_k = R_{k-1} \sqcap M^*_{(k)} = R(t_1,\ldots,t_k)$ holds. ■

COROLLARY 9. Let the function invocation $m(t_1,\ldots,t_n)$ be covered by $\mathcal{M}$. $R(t_1,\ldots,t_n)$ represents the precedence order of all applicable methods (in $\mathcal{M}$) for that invocation. If $R(t_1,\ldots,t_n) \ne \varnothing$, then each factor of $R(t_1,\ldots,t_n)$ includes only one method. ■

We can calculate $R(t_1,\ldots,t_k)$ recursively by referring to (1) and proposition 2(f) as follows:

$$\begin{aligned} R_k &= R_{k-1} \sqcap M^*_{(k)}\left(t_k, T_{(k)}(\mathcal{M})\right) \\ &= R_{k-1} \sqcap M^*_{(k)}\left(t_k, T_{(k)}(\mathcal{M}) \cap T_{(k)}(domain(R_{k-1}))\right) \\ &= R_{k-1} \sqcap M^*_{(k)}\left(t_k, T_{(k)}(domain(R_{k-1}))\right) \qquad (2) \\ &\quad \text{where } k > 0, \text{ and } R_0 = \mathcal{M}. \end{aligned}$$

Now we can discuss mechanisms for constructing and simulating the LUA based on the above results.

5 Constructing the LUA

In this section, we present an algorithm to construct the LUA $\mathfrak{D} = (Q, \Sigma, \delta, q_0, F)$, which depends on the type hierarchy $\mathcal{T}$ and the confusable set $\mathcal{M}$.

The following routine *construct* constructs the LUA through calculating the value of $R(t_1,\ldots,t_k)$ using formula (2). In the following, for each state $q \in Q$ the attribute *q.pord* holds the precedence order at state q.

```
routine construct()
1  {  create a start state q_0
2     q_0.pord ← M_b
3     Q ← Q_0 ← {q_0}
4     for k = 1 to n do
5     {  Q_1 ← ∅
6        foreach q ∈ Q_0 do build_next_states(k, q, Q_1)
7        Q ← Q ∪ Q_1
8        Q_0 ← Q_1
9     }
10    F ← Q_0
11 }
```

In the above routine *construct*, Q_0 represents the set of all states built at level k. The subroutine *build_next_states*(k, q, Q_1) attempts to construct all possible state transitions $\delta(q, t)$ $(t \in \mathcal{T})$ at level k, where the argument Q_1 represents the set of all states built so far at level k+1. After subroutine *build_next_states*(k, q, Q_1) has been executed, Q_1 may be expanded and then represents the new set of all states built so far at level k+1. The following proposition is necessary for the subroutine *build_next_states*.

PROPOSITION 10. Let $m^* \subseteq \mathcal{M}_b$ with $m^* \neq \emptyset$. $\forall\ t \in closure(T_{(k)}(m^*))$ and $1 \leq k < n$, $m^* \cap domain(M^*_{(k)}(t, T_{(k)}(m^*))) \neq \emptyset$ holds.

PROOF. Clearly $closure(T_{(k)}(m^*)) \neq \emptyset$. $\forall\ t \in closure(T_{(k)}(m^*))$, $\exists\ t' \in T_{(k)}(m^*)$ such that $t \preccurlyeq t'$, i.e. $\exists\ m(t_1, \ldots, t_n) \in m^*$ such that $t_k = t'$. The definitions of *domain* and $M^*_{(k)}$ imply that $m(t_1, \ldots, t_n) \in domain(T^{-1}_{(k)}(t')) \subseteq domain(M^*_{(k)}(t, T_{(k)}(m^*)))$, i.e. $m(t_1, \ldots, t_n) \in m^* \cap domain(M^*_{(k)}(t, T_{(k)}(m^*))) \neq \emptyset$. ■

```
subroutine build_next_states(k, q, Q_1)
1  {  T ← T_(k)(domain(q.pord))
2     T' ← closure(T)
3     Σ ← Σ ∪ T'
4     foreach t ∈ T' do
5     {  pord ← q.pord ⊓ M*_(k)(t, T)
6        if k = n then pord ← the first factor of pord
7        if ∃ q' ∈ Q_1 such that q'.pord = pord then
8           δ(q, t) ← q'
9        else
10       {  create a new state q_new and add it to Q_1
11          q_new.pord ← pord
12          δ(q, t) ← q_new
13       }
14    }
15 }
```

Remarks:

(a) Lines 4-14 define a loop, in which a temporal variable, $pord \in \mathfrak{R}$, is introduced to calculate the attribute $\delta(q, t).pord$ and to decide whether $\delta(q, t)$ is a new state. However, the fact $pord \neq \emptyset$ is important so that "the first factor of *pord*" exists, as stated in line 6. This can be proved as follows: we have $pord = q.pord \sqcap M^*_{(k)}(t, T_{(k)}(domain(q.pord)))$, where $t \in closure(T_{(k)}(domain(q.pord)))$ and $1 \leq k < n$. Assume $q.pord \neq \emptyset$, and let $m^* = domain(q.pord)$. Then $m^* \neq \emptyset$. By proposition 10, $\forall\, t \in closure(T_{(k)}(m^*))$, we have $m^* \cap domain(M^*_{(k)}(t, T_{(k)}(m^*))) \neq \emptyset$. By proposition 2(d), we can derive $domain(pord) \neq \emptyset$, i.e. $pord \neq \emptyset$. Therefore, the above fact can be easily proved by induction on k. As a direct result, $\forall\, q \in Q, q.pord \neq \emptyset$. This result will be used in the stage of LUA simulation;

(b) In line 7, we need to compare the equality between $q'.pord$ and $pord$. By theorem 8, $q'.pord = R_k(t'_1, \ldots, t'_k)$ and $pord = R_k(t_1, \ldots, t_k)$, for appropriated types t'_i and t_i ($i = 1, \ldots, k$). Let a be a factor of $q'.pord$, and b a factor of $pord$. By theorem 8(a), either $a = b$ or $a \cap b = \emptyset$ holds.[7] This fact is useful to simplify the implementation of the equality test between $q'.pord$ and $pord$: in order to compare the equality between two sets a and b, we can just compare the smallest elements in a and b.[8]

Let's explain the above algorithms in the context of the example shown in Fig. 1. Suppose that we have already constructed all states and transitions at level 2, where $Q_0 = \{q_3, q_4, q_5, q_6, q_7\}$. Now we are going to construct states and transitions at level 3. Firstly, the subroutine *build_next_states*$(3, q_3, Q_1)$ is invoked, where

- $Q_1 = \emptyset$;
- $q_3.pord = \{m_1\}$, so $T = T_{(3)}(domain(q_3.pord)) = T_{(3)}(\{m_1\}) = \{B\}$, and $T' = closure(T) = \{B\}$;
- In the loop body (lines 5-14), $pord = q_3.pord \sqcap M^*_{(3)}(t, T) = \{m_1\} \sqcap \{m_1, m_2\} = \{m_1\}$, when $t = B$. Therefore, a new state q_8 with precedence order $\{m_1\}$ should be created and then $\delta(q_3, B) = q_8$. Now $Q_1 = \{q_8\}$.

Afterwards, two other subroutine invocations *build_next_states*$(3, q_4, Q_1)$ with $Q_1 = \{q_8\}$ and *build_next_states*$(3, q_5, Q_1)$ with $Q_1 = \{q_8, q_9\}$ are executed. Now Q_1 becomes as $\{q_8, q_9, q_{10}\}$. Consider the next subroutine invocation *build_next_states*$(3, q_6, Q_1)$, where

- $Q_1 = \{q_8, q_9, q_{10}\}$;
- $q_6.pord = \{m_3\}\{m_2\}\{m_1\}$, so $T = T_{(3)}(domain(q_6.pord)) = T_{(3)}(\{m_1, m_2, m_3\}) = \{A, B\}$, and $T' = closure(T) = \{A, B\}$;

---

7. Assume $m \in a \cap b \neq \emptyset$. By theorem 8(a), $a = \{m' \in \mathcal{M}_b \mid m$ and m' have the same first k argument types$\} = b$.

8. We can pre-sort all methods in $\mathcal{M}_b$, so that in each subset of $\mathcal{M}_b$, we can find the smallest element with respect to that order.

- In the loop body (lines 5-14), $pord = q_6.pord \sqcap M^*_{(3)}(t, T)$

$$= \begin{cases} (\{m_3\}\{m_2\}\{m_1\}) \sqcap \{m_3\} = \{m_3\}, & \text{if } t = A; \\ (\{m_3\}\{m_2\}\{m_1\}) \sqcap (\{m_1, m_2\}\{m_3\}) = \{m_3\}\{m_2\}\{m_1\}, & \text{if } t = B. \end{cases}$$

  Therefore, when $t = A$, the equality $pord = \{m_3\} = q_{10}.pord$ yields $\delta(q_6, t) = q_{10}$ (as $q_{10} \in Q_1$); when $t = B$, a new state q_{11} with precedence order $\{m_3\}\{m_2\}\{m_1\}$ is created and then $\delta(q_6, t) = q_{11}$. Now $Q_1 = \{q_8, q_9, q_{10}, q_{11}\}$.

Likewise, the last subroutine invocation at level 3 is *build_next_states*$(3, q_7, Q_1)$ with $Q_1 = \{q_8, q_9, q_{10}, q_{11}\}$, in which a new state q_{12} is created. Consequently, $Q_1 = \{q_8, q_9, q_{10}, q_{11}, q_{12}\}$ is the set of all states constructed at level 3.

6 Simulating the LUA

We have described the approach to construct the LUA for each confusable set. According to theorem 1, a function invocation $m(t_1, \ldots, t_n)$ is covered by at most one confusable set, so that only one LUA should be simulated for that invocation. Furthermore, we know which LUA has to be simulated (we say that this LUA *covers* the function invocation $m(t_1, \ldots, t_n)$). In order to reduce the space-complexity, the LUA does not identify *all* possible dispatching cases. Generally, given a function invocation $m(t_1, \ldots, t_n)$ and a LUA that covers $m(t_1, \ldots, t_n)$, it is not possible to directly employ the standard automaton simulation technique. Rather, a special algorithm to simulate the LUA is necessary where the knowledge of the relation $\preccurlyeq$ is used. In this section, we present the algorithm to simulate the LUA.

6.1 Approach

In the following, let *level*(q) denote the length of the path from state q_0 to state q (i.e. the number of transitions from the start state to q). From the routine *construct*, we can see that *level*(q) is independent of the selected path. For example, in Fig. 1, *level*(q_3) is 2 and there are three different paths from q_0 to q_3. Moreover, we introduce a notion

$$\delta_D(q, \bullet) = \{t \in \Sigma \mid \delta(q, t) \text{ is defined}\}.$$

By the subroutine *build_next_states*, $\forall\, q \in Q - F$, we have

$$\delta_D(q, \bullet) = closure\big(T_{(level(k))}(domain(q.pord))\big).$$

Hence $\delta_D(q, \bullet)$ is closed, for $q \in Q - F$.

THEOREM 11. Let $S \subseteq \mathcal{T}$, $t \in \mathcal{T}$. $\forall\, k$, $1 \le k \le n$, the following holds:

$$M^*_{(k)}(t, S) = \begin{cases} \varnothing, & \text{if } LUB_\diamond(t, closure(S)) = \varnothing; \\ M^*_{(k)}(t', S), & \text{if } LUB_\diamond(t, closure(S)) \neq \varnothing, \text{ and } t' \text{ is the unique element in } LUB_\diamond(t, closure(S)). \end{cases}$$

PROOF. If $LUB_\diamond(t, closure(S)) = \varnothing$, then $\{s \in S \mid t \preccurlyeq s\} = \varnothing$ and hence $M^*_{(k)}(t, S) = \varnothing$. On the other hand, if $LUB_\diamond(t, closure(S)) \neq \varnothing$, by theorem 5, $LUB_\diamond(t, closure(S)) =$

$\{t'\}$. By definition, $M^*_{(k)}(t, S) = \prod_{i=1}^{l} T^{-1}_{(k)}(t_i)$, where $\{t_1, \ldots, t_l\} = \{s \in S \mid t \leqslant s\}$, and $t_i \propto_t t_{i+1}$ for $1 \leq i < l$. Proposition 4 implies that $\{s \in S \mid t \leqslant s\} = \{s \in S \mid t' \leqslant s\}$. Meanwhile by the restriction on the relation $\propto$, $t_i \propto_{t'} t_{i+1}$ still holds for $1 \leq i < p$. This means that $M^*_{(k)}(t', S) = \prod_{i=1}^{l} T^{-1}_{(k)}(t_i)$, i.e. $M^*_{(k)}(t, S) = M^*_{(k)}(t', S)$. ■

Let $m(t_1, \ldots, t_n)$ be a function invocation covered by $\mathcal{M}_\flat$. By theorem 11 and formula (2), we can recursively calculate $R_k = R(t_1, \ldots, t_k)$ (with $R_0 = \mathcal{M}_\flat \in \mathfrak{R}_\flat$) as follows:

$$R_k = R_{k-1} \sqcap M^*_{(k)}(t_k, T_{(k)}(domain(R_{k-1}))) = \begin{cases} \varnothing, & \text{if } LUB_\diamond(t_k, closure(T_{(k)}(domain(R_{k-1})))) = \varnothing; \\ R_{k-1} \sqcap M^*_{(k)}(t, T_{(k)}(domain(R_{k-1}))), & \\ \quad \text{if } \exists \text{ (unique) } t \in LUB_\diamond(t_k, closure(T_{(k)}(domain(R_{k-1})))). & \end{cases} \tag{3}$$

The task of the LUA simulation is to find a final state q such that $q.pord$ is equal to the first factor of R_n. We can accomplish this by tracing a list of states $q_{i_1}, \ldots, q_{i_n}$ with $q_{i_k}.pord = R_k$ (or the first factor of R_k, if $k = n$). According to the formula (3) and the definition of $\delta_D(q, \bullet)$, we have the relation $q_{i_{k+1}} = \delta(q_{i_k}, t)$, where $t \in LUB_\diamond(t_k, \delta_D(q_{i_k}, \bullet))$. Formally,

PROPOSITION 12. Let the LUA $\mathfrak{D} = (Q, \Sigma, \delta, q_0, F)$ cover the function invocation $m(t_1, \ldots, t_n)$. Let $q \in Q - F$ be a state with $q.pord = R_k$, where $k = level(q)$. If $LUB_\diamond(t_{k+1}, \delta_D(q, \bullet)) \neq \varnothing$, then $\delta(q, t).pord = R_{k+1}$ (or its first factor, if $k+1 = n$), for the unique $t \in LUB_\diamond(t_{k+1}, \delta_D(q, \bullet))$.

PROOF. Since $\delta_D(q, \bullet) = closure\big(T_{(k+1)}(domain(q.r))\big)$ is closed for $q \in Q - F$, by theorem 5, $t \in \delta_D(q, \bullet)$ is unique. If $k + 1 < n$, we have

$$\begin{aligned} \delta(q, t).pord &= q.pord \sqcap M^*_{(k+1)}\big(t, T_{(k+1)}(domain(q.pord))\big) \\ &= R_k \sqcap M^*_{(k+1)}\big(t, T_{(k+1)}(domain(q.pord))\big) \\ &= R_{k+1} \qquad \text{(by formula (3)).} \end{aligned}$$

Especially, if $k + 1 = n$, $\delta(q, t).pord$ = the first factor of R_n. ■

The following function *simulate* reads the type-list $t_1, \ldots, t_n$, and then returns either the most specific method (if any exists) in $\mathcal{M}_\flat$ or the predefined error-handling function if there is no applicable method in $\mathcal{M}_\flat$.

```
function simulate()
1  {  q ← q0
2     for k = 1 to n do
3     {  T ← LUB◇(tk, δD(q, •))
4        if T ≠ ∅ then q ← δ(q, t), where t ∈ T
```

```
5           else return the predefined error-handling function
6       }
7       return the (unique) method in the (unique) factor of q.pord
8   }
```

Remarks:

(a) In line 3 the function $LUB_{\diamond}(t_k, \delta_D(q, \bullet))$ is invoked, where $\delta_D(q, \bullet)$ is closed. Under this condition, the implementation of $LUB_{\diamond}$ has been stated in section 4.2;

(b) In the run time environment, the function *simulate* just uses the knowledge of the relation $\preccurlyeq$, the knowledge of the relation $\propto$ is not needed any more;

(c) Line 7 refers to the attribute *q.pord*, which consists of one factor only that includes only one method at the final states (c.f. subroutine *build_next_states*). In fact this is the only place referring to the attribute *q.pord* in the time of simulation. From the view of implementation, it is not necessary to store the attribute *q.pord*, $\forall\, q \in Q - F$, to the run time environment, e.g. in Fig. 1 only the underscored methods at states q_{13}, q_{14}, and q_{15} need to be saved;

(d) According to proposition 12, it is easy to prove the correctness of the function *simulate* by induction.

6.2 Improvement

The time-complexity of function *simulate* depends on the implementation of the function $LUB_{\diamond}$. As stated in Section 4, the worst time-complexity of the function call $LUB_{\diamond}(s, S)$ is $O(\|S\|)$ when S is closed. Consequently, the total time-complexity of function *simulate* will rely on the size of $\delta_D(q, \bullet)$ in line 3 of *simulate*.

In the following we consider an improvement for the time-efficiency of function *simulate*. Let c be a predefined constant independent of any concrete type hierarchy and LUA. During the compile time, $\forall\, q \in \Sigma$, if $\|\delta_D(q, \bullet)\| > c$, we can extend the state transitions $\delta(q, t)$ with respect to the domain of t from $\delta_D(q, \bullet)$ into $\mathcal{T}$ in the following way:

$$\delta(q,t) = \begin{cases} \delta(q,t'), & \text{if } LUB_{\diamond}(t, \delta_D(q,\bullet)) \neq \varnothing, \\ & \text{so } \exists \text{ unique } t' \in LUB_{\diamond}(t, \delta_D(q,\bullet))); \\ q_{-1}, & \text{if } LUB_{\diamond}(t, \delta_D(q,\bullet)) = \varnothing, \\ & \text{where } q_{-1} \text{ is the predefined failure state.} \end{cases}$$

Here, it is necessary to attach an attribute (1-bit) to the state q to indicate whether the corresponding $\delta_D(q, \bullet)$ is extended or not. The following is the improved version of the function *simulate*, after the LUA is extended as above.

```
function simulate() /* improved version */
1   {   q ← q0
2       for k = 1 to n do
3       {   if δD(q, •) is extended then
4           {   q ← δ(q, tk)
5               if q = q-1 then return the predefined error-handling function
```

```
6          }
7          else /* δ_D(q, •) is not extended */
8          {  T ← LUB_◇(t_k, δ_D(q, •))
9             if T ≠ ∅ then q ← δ(q, t), where t ∈ T
10            else return the predefined error-handling function
11         }
12    }
13    return the (unique) method in the (unique) factor of q.pord
14 }
```

Remarks:

(a) In section 4.2, we performed the topological sorting on $\mathcal{T}$, so $\mathcal{T} = \{T[1], T[2], ..., T[\tau]\}$. In the aspect of the LUA representation, $\forall\ q \in \Sigma$, the function $\delta(q, t)$ with respect to the argument $t \in \mathcal{T}$ can be converted into function $f_q(i) = \delta(q, T[i])$ with respect to the number argument i. On the one hand, for $q \in \Sigma$ with $\delta_D(q, \bullet) = \mathcal{T}$ (i.e. the $\delta_D(q, \bullet)$ is extended), the argument domain of f_q is $[1, \tau]$. In this case, f_q can be represented by a linear table so that the time-complexity of f_q is $O(1)$; on the other hand, for $q \in \Sigma$ with $\delta_D(q, \bullet) \neq \mathcal{T}$ (i.e. the $\delta_D(q, \bullet)$ is not extended and $\|\delta_D(q, \bullet)\| \leq c$), the argument domain of f_q is a subset of $[1, \tau]$, with size $\|\delta_D(q, \bullet)\| \leq c$. In this case, f_q can be represented by a hash table or a binary tree. Since c is the predefined constant, the time-complexity of f_q is $O(1)$ also.

A related question is, $\forall\ t \in \mathcal{T}$, how to get a corresponding number i with $t = T[i]$ at run time? As the topological sorting on $\mathcal{T}$ is performed at compile time, each type $t \in \mathcal{T}$ can be identified by the number i, which can be attributed to t. Therefore, $\forall\ t \in \mathcal{T}$, getting the number i such that $t = T[i]$ can be preformed of the time-complexity $O(1)$ at run time.

Conclusively, the time-complexity of the improved function *simulate* is $O(n)$, where n is the method arity;

(b) If all transitions $\delta(q, t)$ are extended for all $q \in Q$, then even the knowledge of relation $\preccurlyeq$ is not needed in run time environment, since lines 8-11 are never executed;

(c) The extended LUA does not create any new state except the predefined failure state $q_{\perp}$;

(d) The value of the predefined constant c influences the number of transitions in the LUA and the efficiency of the simulation: a higher value of c would save space but would make the algorithm run slower; conversely, a lower value of c would yield the opposite effects. More experience is necessary to find a good value.

7 Experimental Results

Having discussed the algorithms of the LUA construction, we can see that the size of the LUA heavily depends on the structure of the type hierarchy $\mathcal{T}$ and the confusable set $\mathcal{M}_{\mathcal{C}}$. These factors make it difficult to properly evaluate the average space-complexity of the LUA. An alternative way to evaluate the LUA's space-complexity is to

$\|\|\mathcal{T}\|\|$	$\|\|\mathcal{M}_b\|\|$	$arity(\mathcal{M}_b)$	$\|\|Q\|\|$	$\|\|\mathcal{F}\|\|$	$\|\|\mathcal{F}\|\| / \|\|Q\|\|$
30	86	2	117	900	7.69
20	49	3	139	6,092	43.83
35	87	3	296	30,019	101.42
21	103	4	613	194,481	317.26
18	73	4	396	103,976	262.57

Table 1. Experimental results of five large applications

investigate several large applications chosen at random. For each application, we calculate the following parameters:

- $\|\mathcal{T}\|$ the number of types in $\mathcal{T}$,
- $\|\mathcal{M}_b\|$ the number of methods in $\mathcal{M}_b$,
- $arity(\mathcal{M}_b)$ the arity of methods in $\mathcal{M}_b$,
- $\|Q\|$ the number of states in the corresponding LUA = $(Q, \Sigma, \delta, q_0, F)$, and
- $\|\mathcal{F}\|$ the number of all possible dispatch cases, where
  $\mathcal{F}$ = {function invocation m covered by $\mathcal{M}_b$ | $\exists\, m' \in \mathcal{M}_b$ such that m' is applicable for m}

Since other lookup structures proposed in [13, 16] needed to prefill all possible dispatch cases $\mathcal{F}$ (we discuss this in section 8), as a comparison point, we calculate the parameters $\|Q\|$, $\|\mathcal{F}\|$, and $\|\mathcal{F}\|/\|Q\|$ for each application. For the example shown in Fig. 1, the values of these parameters are $\|\mathcal{T}\| = 8$, $\|\mathcal{M}_b\| = 3$, $arity(\mathcal{M}_b) = 4$, $\|Q\| = 16$, $\|\mathcal{F}\| = 751$, and $\|\mathcal{F}\|/\|Q\| = 46.94$. We tested five large applications. Table 1 shows these experimental results. Generally, the size of LUA is far less than the size of the other lookup structures [13, 16], i.e. the value $\|Q\|$ is far less than $\|\mathcal{F}\|$. Moreover, it seems that a greater benefit of our LUA technique reveals when the arity of $\mathcal{M}_b$ becomes greater.

8 Related Work

Some related research has been discussed in previous sections. CLOS [6] and its predecessors CommonLoops [5] and Flavor [15] pioneered the use of multi-method dispatching using inheritance order precedence. In the Flavor system, Moon [16] proposed the lookup structure which is organized as a set of hash tables, in which all possible dispatch cases are prefilled. In the presence of multi-methods, however, the number of all possible cases is combinatorially explosive. For a generic function with n arguments, the number of possible cases is in the order of e^n as indicated in [11] (c.f. Table 1). Rodriguez, *et al.* [13] introduced another technique for multiple-dispatching running in CLOS. Similarly, this approach requires to prefill all possible dispatch cases. As a comparison, Dussud [11] suggested a dynamic cache technique to overcome this drawback. The dynamic cache requires memory only for called methods. In the time of dispatching, the search begins in the cache. If the selector has no entry, a

dynamic lookup provides the method procedure address. This address is then stored in the cache. Cache filling is then randomly distributed at execution. That overhead may not be reasonable for real-time systems. However, saved memory is great in large systems where execution deals with few selectors. Essentially, all these techniques are based on hash functions over all argument types, but the problem of combinatorial explosion is not resolved.

Agrawal, *et al.* [2] presented a basis for introducing multi-methods in languages with static type checking and for designing new object-oriented paradigms based on multi-methods. In the time of type checking, they suggested a strategy to find the most specific applicable method, by introducing a *method precedence graph* for a confusable set. If this technique was applied to dynamic dispatching, the space required would be less than that in our lookup automaton, but the time-complexity would be $O(\|\mathcal{M}_b\| \times n)$, where n is the method arity. For large systems, the number $\|\mathcal{M}_b\|$ can reach to the order of 10^2. Consequently, this strategy would be very expensive when extended to dynamic lookup for large systems.

Lécluse and Richard [14] characterized multiple-dispatching in terms of structural subtyping. Whenever two confusable methods are not ordered by argument subtype precedence, they require that additional methods are defined to insure that a most specific applicable method for any given set of arguments can always be determined by the use of argument subtype precedence alone. Mugridge, *et al.* [17] discussed multi-method dispatch with static type checking. They described a method-specificity rule that only partially orders the methods of a generic function. They defined the *cover* of a method $m(t_1,\ldots,t_n)$ as the cross-product of all possible argument types: $\{(s_1,\ldots,s_n) \mid \forall\, i,\ s_i \preccurlyeq t_i\}$. Method applicability is defined in terms of the non-empty intersection of the covers of the function call and the method definitions. Given a call, applicable methods are found by intersecting corresponding covers. Since the method precedence rule does not totally order methods, if two applicable methods are found that do not have an order defined between them, such a call is declared ambiguous.

Finally, in the presence of single-dispatching, Dixon [9] and André, *et al.* [4] applied the minimal coloring theories to construct lookup tables. In single-dispatching, only the types of receivers are of interest. Although all possible dispatch cases are prefilled in the lookup tables, yet the memory overhead may not be so crucial than in multiple-dispatching. On the other hand, Ingalls [12] introduced a simple approach to simulate multiple-dispatching in a system that just supports single-dispatching. Single-dispatching can be applied to each interesting argument in turn, to simulate the effect of dispatching on all interesting arguments. However, this approach may result in a combinatorially explosive situation, such that even the minimal coloring techniques are still powerless to reduce the space-complexity of the lookup tables.

9 Conclusions and Discussions

We have presented a mechanism for implementing multi-method dynamic lookup based on the lookup automaton (LUA) technique. For a given type hierarchy and a method confusable set, the corresponding LUA is constructed at compile time and is

simulated at run time. Several characteristics of this dynamic lookup strategy can be concluded as follows:

- The dynamic lookup is transformed into the LUA simulation. For each generic function invocation, we trace the precedence order $R(t_1,\ldots,t_n)$ on a list of the corresponding states of the LUA. The most specific applicable method (if any exist) can be found in the final state;
- For any function invocation, the number of state transitions is not greater than n, where n is the function arity. The total time-complexity for the LUA simulation is $O(n)$;
- In the run time environment, the knowledge of the relations $\propto$ is not needed. The LUA simulation can be performed solely on the knowledge of the relation $\preccurlyeq$. In some special cases, even the knowledge of relation $\preccurlyeq$ is obsolete;
- The size of the LUA heavily depends on the structure of the type hierarchy and the confusable set. In general, the size is far less than the current cache techniques [13, 16]. This is because in LUA the states (at the same level) can be "merged" together if they have the same precedence order. Our experience supports this claim;
- The procedure to reduce the size of the LUA is performed at the time of constructing the LUA. The advantage of this approach is that a large memory overhead can be prevented both at compile time and at run time.

We assume that, once the structure of the type hierarchy or the confusable set have been updated, the LUA must be rebuilt, even in the case that only one method is added or removed. In this case, one particular question is whether it is possible to update the old LUA rather than to compute it again from scratch. To answer this, it might be necessary to look at the dynamic behavior of the *closure* function, e.g. how to calculate $closure(S-\{x\})$ or $closure(S \cup \{x\})$ from $closure(S)$. To determine the LUA's dynamic behavior, more research and experience are needed.

Acknowledgements. We thank Klemens Böhm, Peter Muth, Gisela Fischer and Karl Aberer for their comments on an earlier version of this paper. Thoughtful referee comments are gratefully acknowledged.

References

1. AGRAWAL, R., BORGIDA, A., AND JAGSDISH, H. V. Efficient Management of Transitive Relationships in Large Data And Knowledge Bases. In *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, 1989.
2. AGRAWAL, R., DEMICHIEL, L. G., AND LINDSAY, B. G. Static Type Checking of Multi-Methods. In *Proc. Conf. OOPSLA*, 1991.
3. AHO, A. V., AND ULLMAN. J. D. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, INC. 1972.
4. ANDRÉ, P., AND ROYER, J.-C. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *Proc. Conf. OOPSLA*, 1992.

5. Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. CommonLoops: Merging Lisp and Object-Oriented Programming. In *Proc. Conf. OOPSLA*, 1986.

6. Bobrow D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. Common Lisp Object System Specification X3J13. In *SIGPLAN Notice* 23, *special issue*, Sept. 1988.

7. Cardelli, L., and Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4), Dec. 1985.

8. Chambers, C. Object-oriented Multi-Methods in Cecil. In *Proc. Conf. ECOOP*, 1991.

9. Dixon, R. Vaughan, P., and Schweizer, P. A Fast Method Dispatcher for Compiled Language with Multiple Inheritance. In *Proc. Conf. OOPSLA*, 1989.

10. Ducournau, R., Habib, M., Huchard, M., and Mugnier, M. L. Monotonic Conflict Resolution Mechanisms for Inheritance. In *Proc. Conf. OOPSLA*, 1992.

11. Dussud, P. H. TICLOS: An Implementation of CLOS for the Explorer Family. In *Proc. Conf. OOPSLA*, 1990.

12. Ingalls, D. H. H. A Simple Technique for Handing Multiple Polymorphism. In *Proc. Conf. OOPSLA*, 1986.

13. Kiczales, G., and Rodriguez, L. Efficient Method Dispatch in PCL. In *Proc. Conf. on Lisp and Functional Programming*, 1990.

14. Lécluse, C., and Richard, P. Manipulation of Structured Values on Object-Oriented Databases. In *Proc. Second Int'l Workshop on Database Prog. Lang.*, 1989.

15. Moon, D, and Weinreb, D. *Lisp Machine Manual*, MIT AI Lab, 1981, Chapter 20.

16. Moon, D. Object Oriented Programming with Flavors. In *Proc. Conf. OOPSLA*, 1986.

17. Mugridge, W. G., Hamer, and Hosking, J. G. Multi-Methods in a Statically-Typed Programming Language. In *Proc. Conf. ECOOP*, 1991.

Appendix

In the following the proofs of lemma 6 and theorem 7 are presented.

Lemma 6:

Proof. We prove (a) and (b) by induction on k from τ down to 2. The case $k = \tau$ is obvious. Assume that the lemma holds for all k with $l < k \leq \tau$. We prove that the lemma holds when $k = l > 1$.

(a) Let $a, b \in \bar{T}_k$. Clearly $a \prec_k b$ implies $a \prec_{k+1} b$, so that, by the assumption of the induction, $a \prec b$ holds. It remains to prove that $a \prec b$ implies $a \prec_k b$. Let $a = T[i_1]$ and $b = T[i_2]$. The assumption of the induction is $a \prec_{k+1} b$. If $\mu[k] = in\_closure$, then

$\mathcal{T}_k = \mathcal{T}_{k+1}$ so that $a \prec_k b$ holds; if $\mu[k] = not\_in\_closure$, then $\mathcal{T}_k \neq \mathcal{T}_{k+1}$. Since $a \prec b$, there must be a path $\mathcal{P}$ from a to b in $\mathcal{T}$. If $T[k]$ is not in $\mathcal{P}$, then the relation $a \prec_k b$ must conform with $a \prec_{k+1} b$; if $T[k]$ is in $\mathcal{P}$, then $T[i_1] \preccurlyeq T[k] \preccurlyeq T[i_2]$ holds. Since $T[k] \notin \tilde{T}_k$ and $T[i_{1,2}] \in \tilde{T}_k$, the relation $T[i_1] \prec T[k] \prec T[i_2]$ must hold such that $i_1 < k < i_2$ and $\mu[i_2] = in\_closure$. Hence, by assumption (a), $T[i_1] \prec_{k+1} T[k] \prec_{k+1} T[i_2]$ and $\mu[i_2] = in\_closure$ hold in $\tilde{T}_{k+1}$. By assumption (b), in $\tilde{T}_{k+1}$, $\exists\, i'_2$, $k+1 \leq i'_2 \leq i_2$, such that $T[k] \vec{\prec}_{k+1} T[i'_2] \preccurlyeq_{k+1} T[i_2]$. In $\tilde{T}_k$, $\mu[k] = not\_in\_closure$ holds, i.e. when the loop-variable i is equal to k, lines 10-11 are executed, where $T[j_{min}] \preccurlyeq_{k+1} T[i'_2]$ holds. Moreover, $T[j_{min}] \preccurlyeq_k T[i'_2] \preccurlyeq_k T[i_2]$ must also hold in $\tilde{T}_k$, so that $T[i_1] \prec_k T[j_{min}]$ yields $T[i_1] \prec_k T[i_2]$.

(b) By (a), we know that the relation $T[k-1] \prec_k T[j]$ (in $\mathcal{T}_k$) implies $T[k-1] \prec T[j]$ (in $\mathcal{T}$). If $T[k-1] \vec{\prec} T[j]$, then $T[k-1] \vec{\prec}_k T[j]$ holds so that $j' = j$ is the result; otherwise $\exists$ a path $\mathcal{P}$ from $T[k-1]$ to $T[j]$ in $\mathcal{T}$, and $\exists\, l$, $k-1 < l \leq j$, such that $T[k-1] \vec{\prec} T[l] \prec T[j]$. If $\mu[l] = in\_closure$, then $j' = l$ is the result; otherwise we consider the hierarchy $\mathcal{T}_{l+1}$. By the assumption of the induction, $T[l] \prec_{l+1} T[j]$ holds, while $\exists\, j'$, $l+1 \leq l' \leq j$, such that $T[l] \vec{\prec}_{l+1} T[l'] \preccurlyeq_{l+1} T[j]$. In $\mathcal{T}_l$, $\mu[l] = not\_in\_closure$, i.e. when the loop-variable i is equal to l, lines 10-11 are executed, where $T[j_{min}] \preccurlyeq_{l+1} T[l']$. Hence, $T[k-1] \vec{\prec}_l T[j_{min}]$ (updated at line 11). This relation must also be kept in $\mathcal{T}_k$, i.e. $T[k-1] \vec{\prec}_k T[j_{min}]$. By (a), $T[j_{min}] \preccurlyeq_{l+1} T[l']$ incurs $T[j_{min}] \preccurlyeq T[l']$ so that $T[j_{min}] \preccurlyeq_k T[l']$ holds. Hence, $j' = j_{min}$ is the result.

It is still necessary to prove (a) for the case $k = 1$. Since $\mathcal{T}_1 = \mathcal{T}_2$ and $\tilde{T}_1 \subseteq \tilde{T}_2$ hold, so the relationship $\prec_1$ conforms with $\prec_2$ in $\tilde{T}_1$. By the assumption of the induction, the relationship $\prec_2$ conforms with $\prec$ in $\tilde{T}_2$, so that $\prec_1$ conforms with $\prec$ in $\tilde{T}_1$. ■

THEOREM 7:

PROOF. Let $S' = \{T[j] \in \tilde{T}_1 \mid \mu[j] = in\_closure,\ 1 \leq j \leq \tau\}$. Clearly $S \subseteq S'$ and it suffices to prove that S' is closed and $S' \subseteq closure(S)$.

Let $T[i_1]$, $T[i_2] \in S'$ and $T[j] \in GLB(T[i_1], T[i_2])$. Assume $T[j] \notin S'$. Then neither $T[i_1] \preccurlyeq T[i_2]$ nor $T[i_2] \preccurlyeq T[i_1]$ holds. Hence $j < min(i_1, i_2)$. In $\mathcal{T}_{j+1}$ we have $\mu[i_{1,2}] = in\_closure$. By lemma 6, $T[j] \prec_{j+1} T[i_{1,2}]$ holds and $\exists\, i'_1$ and i'_2, such that $T[j] \vec{\prec}_{j+1} T[i'_l] \preccurlyeq_{j+1} T[i_l]$, for $l = 1, 2$. Clearly $i'_1 \neq i'_2$ holds (otherwise $T[j] \notin GLB(T[i_1], T[i_2])$). Consider the case that the loop-variable i is equal to j. For the variable j_{min} (c.f. line 6) we have $T[j] \vec{\prec}_{j+1} T[j_{min}] \preccurlyeq_{j+1} T[i'_l] \preccurlyeq_{j+1} T[i_l]$ and $T[j_{min}] \in S'$, for $l = 1, 2$. But this implies $T[j] \notin GLB(T[i_1], T[i_2])$. This is a contradiction and hence S' is closed.

Next, we prove $S' \subseteq closure(S)$, i.e. $\forall\, k$, $1 \leq k \leq \tau$, if $\mu[k] = in\_closure$, then $T[k] \in closure(S)$. The proof is by induction on k from τ down to 1. The case $k = \tau$ is obvious. Assume that it is true for all k with $l < k \leq \tau$. Consider the case $k = l \geq 1$. When the loop-variable i is equal to k, then since $\mu[k] = in\_closure$, line 8 is executed. Thus, $\exists\, j \in \Delta$, $j \neq j_{min}$ such that $\mu[j] = \mu[j_{min}] = in\_closure$ and $T[j_{min}] \prec_{k+1} T[j]$ does not hold in $\mathcal{T}_{k+1}$. Now it suffices to prove that $T[k] \in GLB(T[j], T[j_{min}])$. Clearly $T[j] \prec_{k+1} T[j_{min}]$ does not hold, i.e. $T[j]$ and $T[j_{min}]$ are not comparable with respect to the relation $\preccurlyeq_{k+1}$

in $\mathcal{T}_{k+1}$. Suppose that $T[k] \notin GLB(T[j], T[j_{min}])$. By proposition 3, $\exists\ k'$, $k < k' < min(j, j_{min})$, such that $T[k] \prec_{k+1} T[k'] \in GLB(T[j], T[j_{min}])$. Since S' is closed, $T[k'] \in S'$ holds, i.e. $\mu[k'] = in\_closure$ in $\mathcal{T}_{k+1}$. By lemma 6(b), in $\mathcal{T}_{k+1}$, $\exists\ k''$, $k'' \leq k'$, such that $T[k] \vec{\preceq}_{k+1} T[k''] \preccurlyeq_{k+1} T[k']$ and $\mu[k''] = in\_closure$. Therefore, $k'' \leq k' < j_{min}$ holds. But by the definition of j_{min} (c.f. line 6), we have $j_{min} \leq k''$. This contradiction completes the proof. ■