5.5 Dynamic dispatching

In the incremental version of the dispatching problem we must support insertions of new types as leaves (with their accompanied method implementations), while answering dispatching queries. More precisely, the problem is to maintain a data structure to manage a hierarchy of types annotated with message families, which supports two kinds of operations:

(1) dispatch(μ, t), where t is a type and μ is a message.

(2) Insert$(t, t_1, \ldots, t_p, \mu_1, \ldots, \mu_q)$ which is an insertion of new type t into the hierarchy. Types $t_1, \ldots, t_p$, $p \geq 0$ are the parents of t, and must have been previously inserted. Type t also implements messages $\mu_1, \ldots, \mu_q$, $q > 0$. These messages may already been introduced by a previous type, or they might be new.

This section describes two algorithms for generalizing the static dispatching algorithm to support such insertions. We use the same slicing technique as before, where a dispatch(μ, t) query searches for t in the data structure of μ corresponding to the (previously computed) slice of t. In inserting a new type, we shall use order-preserving heuristic (see Section 5.2) for maintaining the slicing property; in other words, an insertion of a new type will never disturb the ordering of types in each slice. The challenge is to efficiently update the data structure of each message in each of the slices. Slicing and the implementation of a dispatch query guarantee that updates done in different slices are entirely independent.

We now describe how the problem of incremental dispatching can be solved using κ dictionaries over the ordered lists of the κ slices (see Section 3.2). Consider some fixed slice $\mathcal{T}_i$. Recall that all the types in $\mathcal{T}_i$ are kept in an ordered list such that the following slicing property holds: *for each type $t \in \mathcal{T}$, the set of descendants in slice $\mathcal{T}_i$, denoted $D_i(t)$, defines a list interval* (which might be empty). The order-preserving heuristic selects a unique slice $\mathcal{T}_i$, and a location in it where t can be inserted without disturbing the slicing property. With a slight abuse of notation we will use $\mathcal{T}_i$ to denote the entire ordered list, and $D_i(t)$ to denote its list interval. No confusion will arise.

Each method $\mu(t) \in F_\mu$ defines an interval $D_i(t)$ in the list $\mathcal{T}_i$. The collection of intervals of F_μ partition the list $\mathcal{T}_i$ into *list segments*. Dispatching μ on types in the same segment results in the same method to execute.

Let $\Upsilon_\mu[i]$ denote the dictionary of μ over the list $\mathcal{T}_i$. A dispatch(μ, t) query is translated to a Lookup(t) query in $\Upsilon_\mu[s_t]$, where s_t is the slice of t.

An Insert$(t, t_1, \ldots, t_p, \mu_1, \ldots, \mu_q)$ transaction is handled by first using the order-preserving heuristic to find a slice $\mathcal{T}_i$ and a location ℓ in the ordered list of $\mathcal{T}_i$ where we can insert t without disturbing the slicing property.

Let $M(t)$ be the set of messages recognized by t. Next we need to update the dictionaries $\Upsilon_\mu[i]$, for each $\mu \in M(t)$. The parameters passed to the Update transaction are the location ℓ and the dispatching result dispatch(μ, t). Calculating dispatch(μ, t) can be done at compile time. In general, we assume a preprocessing step (performed in compile-time) in which we can compute any information which depends solely on ancestors of t (no knowledge of future descendants is used).

By using the non-exhaustive dictionary implementation (see Section 3.3) the time of inserting a type t which recognizes a set of messages $M(t)$ is

$$O\left(\sum_{\mu \in M(t)} \log|F_\mu|\right). \tag{20}$$

Therefore, the amortized cost of inserting a method $\mu(t)$ is $O(\Delta_{\mu(t)} \log|F_\mu|)$ time, where $\Delta_{\mu(t)}$ is the size of the cone of influence of $\mu(t)$.

If we use a partitioning dictionary (see Section 3.6) then we must also supply the set of intervals $N \subseteq \text{neighbors}(\ell)$ which contain t. We first calculate the sets $X_\mu = \text{ancestors}(t) \cap F_\mu$ by traversing the ancestors of t. Then N are the list intervals $D_i(t')$ where $t' \in X_\mu \cap \text{neighbors}(\ell)$. Observe that $|N| \leq |\text{ancestors}(t) \cap F_\mu|$. Next we create new intervals for those $D_i(t'), t' \in X_\mu$ which were previously empty.

The time of *all* insert transaction for a message μ, $F_\mu \subseteq \mathcal{T}$, in slice $\mathcal{T}_i$ is

$$f_\mu \log f_\mu + \sum_{t \in \mathcal{T}_i} |\text{ancestors}(t) \cap F_\mu|.$$

Since we apply this algorithm in each of the κ slices, the total time is

$$\kappa f_\mu \log f_\mu + \sum_{t \in \mathcal{T}} |\text{ancestors}(t) \cap F_\mu|,$$

and using simple algebraic manipulation we get

$$\kappa f_\mu \log f_\mu + \sum_{t \in F_\mu} |\text{descendants}(t)|.$$

Therefore, the amortized time of inserting $\mu(t)$ is

$$O(\kappa \log f_\mu + |\text{descendants}(t)|).$$

6. DATA SET

Forty-three hierarchies collected from eight different programming languages and totaling 78,474 types, comprise our experimental data set.

For benchmarking subtyping algorithms we used the same 13 hierarchies for PQE benchmark [Zibin and Gil 2001]. This collection includes all the hierarchies used in previous experimental work on subtyping. As observed previously [Eckel and Gil 2000] many of the topological properties of these hierarchies are similar to those of balanced binary trees. We note that the average number of ancestors in these hierarchies is less than 9 for all hierarchies, with the exception of Geode (14.0) and Self (30.9).

We were unable to obtain information on the definition of messages and methods in eight hierarchies out of our subtyping benchmark. Instead, the data set for benchmarking the dispatching algorithms was assembled from the following sources:

(1) The four hierarchies (Self, Unidraw, LOV, Geode) used in benchmark of RD in MI hierarchies [Driesen and Hölzle 1995]. (These hierarchies were also used in benchmarking the subtyping algorithms.)

(2) The eight SMALLTALK, OBJECTIVE-C and C++ hierarchies used for benchmarking RD and CT [Vitek and Horspool 1996] in SI hierarchies.