EFFICIENT ALGORITHMS FOR THE RUNTIME ENVIRONMENT OF OBJECT ORIENTED LANGUAGES

RESEARCH THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

YOAV ZIBIN

SUBMITTED TO THE SENATE OF THE TECHNION—ISRAEL INSTITUTE OF TECHNOLOGY

SHEBAT, 5764 HAIFA FEBRUARY, 2004

THIS RESEARCH THESIS WAS SUPERVISED BY DR. YOSSI GIL UNDER THE AUSPICES OF THE COMPUTER SCIENCE DEPARTMENT

ACKNOWLEDGMENT

Thank you Eva for keeping me alive

THE GENEROUS FINANCIAL HELP OF THE TECHNION IS GRATEFULLY ACKNOWLEDGED

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

The *object oriented* (OO) paradigm has become the norm for software development. OO languages, such as C++ [124], JAVA [7], EIFFEL [97], and SMALLTALK [71], are used in almost every software project. The OO programming style, and the languages that enable it, have acquired an aura of respectability. OO programming promotes reusability, extendibility, reliability, and portability. All these blessings come however at a *cost of runtime efficiency*. Better understanding of this cost, and finding ways to reduce it, are the subject of this thesis.

This thesis presents our contributions to the following three fundamental problems in the runtime environment of OO languages: subtyping tests [136], message dispatching (both single and multiple dispatching) [137, 138], and object layout [139]. It is important to note that although the problems take variations in different languages, these variations are minor in the implementation of these languages. Out results will therefore be of general interest, and applicable to many different languages.

The thesis also includes an application of the newly developed techniques which enabled us to develop the best algorithm for deciding isomorphism of simple types [140], i.e., whether two non-recursive types using product- and function-type constructors, are isomorphic under the axioms of commutative and associative products, and currying and distributivity of functions over products. In particular, we show that this problem can be solved in $O(n \log^2 n)$ time and $O(n)$ space, where n is the input size. This result improves upon the $O(n^2 \log n)$ time and $O(n^2)$ space bounds of the best previous algorithm. We also describe an $O(n)$ time algorithm for the *linear isomorphism problem*, which does not include the distributive axiom, thereby improving upon the $O(n \log n)$ time of the best previous algorithm for this problem.

The above contributions were published in five conference papers [136–140], two of which [136, 140] were accepted to a journal [69, 70].

List of Abbreviations

Acronym	Expansion
BPE	Bit Packed Encoding (subtyping technique)
BTS	Bit Type Slicing (subtyping and dispatching technique)
CNT	Compressed N-dimensional Tables (multi-dispatching technique)
CPQE	Coalesced PQ-Encoding (subtyping technique)
CT	Compact dispatch Tables (dispatching technique)
DAG	Directed Acyclic Graph
DFS	Depth First Search (traversal of a DAG)
DPH	Dynamic Perfect Hashing
JDK	Java Development Kit
LDT	Layers Dispatch Table (used in TDBD layout)
MI	Multiple Inheritance
NHE	Near Optimal Hierarchical Encoding (subtyping technique)
OO	Object Oriented
PE	Packed Encoding (subtyping technique)
PQE	PQ-Encoding (subtyping technique)
RD	Row Displacement (dispatching technique)
SC	Selector Coloring (dispatching technique)
SI	Single Inheritance
SRP	Single-Receiver Projections (multi-dispatching technique)
TDBD	Two-Dimensional Bi-Directional (object layout technique)
TS	Type Slicing (subtyping and dispatching technique)
VBPTR	Virtual Base Pointer (in C++)
VFT	Virtual Function Tables (dispatching technique)
VPTR	Virtual Function Pointer (a pointer to a virtual function table in C++)

List of Symbols

The following table explains the meaning of symbols used throughout our papers. The next tables explain symbols used only in a single paper: PQ-tree symbols used in Chapter 3, Object layout symbols used in Chapter 6, and Isomorphism symbols used in Chapter 7.

Symbol	Denotes
$\mathcal{F}$	The set of families; $\mathcal{F} \subseteq \wp(\mathcal{T})$
$\mathcal{T}$	The set of types
$\mathcal{T}'$	The *multiple inheritance core* of the hierarchy, i.e., types which have a descendant with more than one parent
$\mathcal{T}^c$	The set of multi-types, i.e., tuples of c types
$\mathcal{T}_i$	The set of types in slice i
β	The number of bits in a bitvector encoding, e.g., NHE
ℓ	Sum of cardinalities of all families, or the number of different method implementations; $\ell \leq w$
ι	The maximal compression factor over the dispatching matrix; $\iota = (nm)/\ell$
κ	The complexity of a hierarchy, or the number of slices used by the TS technique
$D_i(t)$	The descendants of t in slice $\mathcal{T}_i$
F	A set of types defining the same message; $F \in \mathcal{F}$
S	A slice, i.e., a subset of types; $S \subseteq \mathcal{T}$
$T_i(F)$	The set of all types which occur in position i in some tuple of the multi-family F
$a, b, t, t', t"$	Type variables; $a, b, t, t', t" \in \mathcal{T}$
c	An arity of a multi-method
k	The number of slices in CT, SC, (B)PE, (C)PQE
m	Number of families; $m = \|\mathcal{F}\|$
n	Number of types; $n = \|\mathcal{T}\|$
o	Some object
s_t	The slice of type t
w	Number of non-null entries in the dispatching matrix; $\ell \leq w \leq nm$
x, y	Some integers in the range $[1, \ldots, n]$
id_t	The position of type t in the ordering of its slice

Symbol	Denotes
$\mathrm{level}(t)$	the length in nodes of the longest directed (upgoing) path starting from type t
$\min(X)$	The subset of smallest types in X
$\mathrm{pred}(x)$	The immediate predecessor of x in some given set of integers
vec_a	The bit-vector of type a in NHE
$\mathsf{A}, \ldots, \mathsf{J}$	Concrete types
$\mathsf{a}, \ldots, \mathsf{m}$	Concrete messages
$\mathsf{cand}(F, t)$	The ancestors of t which are members of F
$\mathsf{dispatch}(F, t)$	The result of dispatching type t over family F
$\mathsf{g\text{-}dispatch}(F, t)$	The result of a generalized dispatching query: *the entire set* of smallest candidates
$(t_1, \ldots, t_c)$	A multi-type in $\mathcal{T}^c$
$[l_t, r_t]$	The interval of ids of descendants of t in relative numbering (in SI hierarchies)
$\preceq$	The subtype relation (the transitive closure of $\prec_{\mathrm{d}}$)
$\prec_{\mathrm{d}}$	The direct subtype relation (the transitive reduction of $\preceq$)
$\top$	The root of a SI hierarchy

PQ-tree symbols used in Chapter 3

Symbol	Denotes
$\mathcal{P}$	A PQ-tree
$\mathcal{P}^\top$	The universal PQ-tree, which represents all possible orderings
$\mathrm{consistent}(\mathcal{P})$	The set of orderings represented by $\mathcal{P}$
$\perp$	A PQ-tree which represents an empty set of orderings

Object layout symbols used in Chapter 6

Symbol	Denotes
$\Delta_{\mathtt{f}}$	The offset of a field `f` within its type
Δ_t	The offset of type t
ℓ_t	The layer of type t
$\theta_t(i)$	The number of ancestors of type t in layer i
θ_t	The number of ancestors of type t
$A_t(k)$	The expected number of extra dereferences required to access k random fields in t
L	The number of layers in TDBD; $L = \lceil s/2 \rceil$
L_t	The number of non-empty layers in type t
s	The number of colors required to color the conflict graph, which is also the number of semi-layers

Isomorphism symbols used in Chapter 7

Symbol	Denotes
$\mathbf{P}_\perp$	A dummy $\mathbf{P}$-node without a `parent` edge and without terms $\varphi(\mathbf{P}_\perp) = \emptyset$
$\mathcal{T}$	The product tree: Nodes are the $\mathbf{P}$-nodes of the $\mathbf{P}/\mathbf{F}$-graph, edges are defined by `parent` pointers.
$\phi(v)$	The union of terms of ancestors of v in the product tree
ρ, ρ'	Types conforming to the product grammar
σ, σ'	Types conforming to the no-unit grammar
τ, τ'	Types conforming to the general grammar
$\varphi(v)$	The set of term nodes v
$\mathbf{P}/\mathbf{F}$-graph	A graph whose nodes are either $\mathbf{P}$-nodes or $\mathbf{F}$-nodes, and that `parent` edges make a tree $\mathcal{T}$
$h(\tau)$	The length of the longest path from τ to any leaf
u	An $\mathbf{F}$-node (a node representing a function type with an argument- and return-type)
v	A $\mathbf{P}$-node (a node representing a product type with a set of term nodes)
`arg`(u)	A $\mathbf{P}$-node which is the argument type of u
`parent`(v)	A $\mathbf{P}$-node, from which v inherits additional terms
`ret`(u)	A primitive-type specifying the return type of u
$\bowtie$	An operation which concatenates of the terms of two products

Chapter 1

Introduction

The *object oriented* (OO) programming paradigm has become the norm for software development. OO languages, such as C++ [124], JAVA [7], EIFFEL [97], and SMALLTALK [71], are used today in almost every software project, since it is widely recognized that the paradigm promotes reusability, extendibility, reliability, and portability. All these blessings come however at a *cost of runtime efficiency*. Better understanding of this cost, and finding ways to reduce it, are the subject of this thesis.

A unique challenge in the implementation of OO languages is providing support for runtime operations related to the *type hierarchy*. Formally, a hierarchy is a partially ordered set ($\mathcal{T}$,$\preceq$) where $\mathcal{T}$ is a set of *types* and $\preceq$ is a reflexive, transitive and anti-symmetric *subtype relation*. The distinction between type, class, interface, signature, etc., as it may occur in various programming languages does not concern us here. We shall refer to all these collectively as types. If a and b are types, and $a \preceq b$ holds, we say that a is a *subtype* of b and that b is a *supertype* of a. For example, in Figure 1.1, Polygon has three subtypes: Rectangle, Triangle, and Polygon itself.

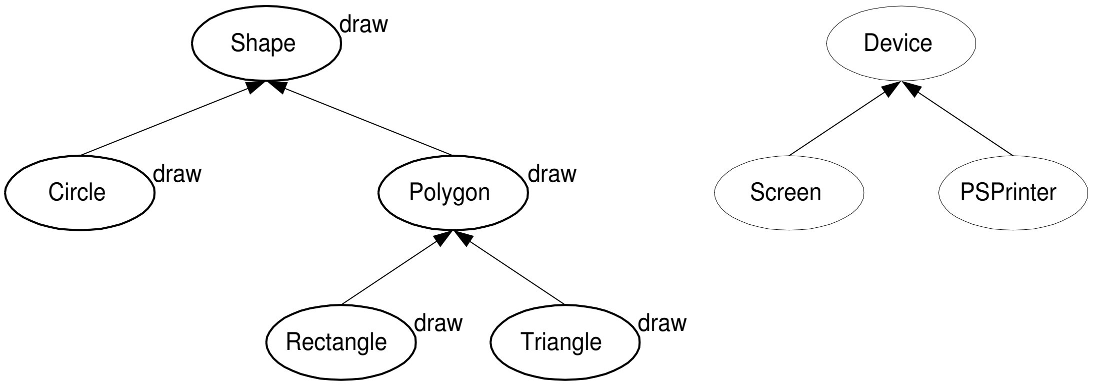


Figure 1.1: An example of a small hierarchy

The most important property derived from a type hierarchy is *type conformance*, meaning that code applicable for type b is also applicable for any of its subtypes a, $a \preceq b$. Cook [33] defines type conformance as

“*a relation intended to capture the notion of one type being immediately com-*

> *patible with another, in a sense that in a context where a value of some type is expected, any value of a conforming type can be used.*"

For example, with the hierarchy of Figure 1.1, code expecting a Shape can receive any of its subtypes, e.g., a Rectangle or a Circle. We therefore have a shape entity that at runtime can refer to any subtype of Shape. Such phenomena are known as *inclusion polymorphism* [23]. Meyer's [96, Sect. 10.1.5 Polymorphism] definition of inclusion polymorphism is:

> "Polymorphism *means the ability to take several forms. In object-oriented programming, this refers to the ability of an entity to refer at run-time to instances of various classes.*"

The consequences of inclusion polymorphism are that although the exact type of a certain entity *must* be known at *runtime*, this type *cannot* be predicted at *compile time*. This polymorphism is the major source of inefficiencies in OO programs. For instance, what happens if we instruct a shape to draw itself? Since different shape entities have different drawing methods, we need to execute the drawing method corresponding to the dynamic type of the shape.

In this thesis, I focused on four ensuing algorithmic problems in the implementation of the runtime environment of OO languages:

1. **Subtyping tests** Given an object o and a type b, a subtype test is to determine whether a, the type of o, is a subtype of b, $a \preceq b$. Such tests (also known as *type inclusion* tests), occur either implicitly in type cast operations, e.g., **`dynamic_cast`** in C++ [124], **`?=`** in EIFFEL [97], or explicitly in the execution of dedicated lingual constructs such as JAVA's [7] **`instanceof`**, and SMALLTALK's [71] `isKindOf:` method. Although there are many efficiency metrics to the problem, in general, we seek an implementation which simultaneously optimizes the space consumption of the hierarchy representation and the time for executing these tests in runtime. Subtyping tests have enjoyed extensive attention (see e.g., [1, 22, 25, 58, 59, 74, 75, 84, 90, 115, 127, 133]).

   A very difficult and interesting (but somewhat language specific) variant of the subtyping problem occurs in EIFFEL (and to a lesser extent in the arrays of JAVA). In this variant the type hierarchy is compounded by an interplay with genericity. For example, if a double ended queue (DQueue) is a subtype of Queue, then a DQueue of Rectangle is a subtype a Queue of Polygon.

2. **Single dispatching** *Message dispatching* stands at the heart of object-oriented (OO) programs, being the only way objects communicate with each other. A *dispatching query* finds the appropriate implementation of the message to be called, according to the dynamic type of the message receiver.

   Driesen and Hölzle [48] found C++ programs spend a median of 5.2% of their time and 3.7% of their instructions in dispatch code. (This is only the direct cost; There is also the cost of lost optimizations opportunities, such as inlining.) Furthermore, it is customary to see dynamically typed languages spend more than 20% of their time dispatching messages [130]. The dispatching problem has been studied extensively (see e.g., [31, 37, 44–50, 78, 86, 100, 102, 118, 130–132, 135]), the main objective functions being the space requirement and dispatching time.

3. **Multiple dispatching** In single dispatching, the method to invoke depends only on the type of a single argument, namely, the receiver. Sometimes it is necessary to dispatch over several arguments, which is then called *multiple dispatching*.

   For example, on the right of Figure 1.1, we see a hierarchy of rendering devices, such as a Screen and PSPrinter. A shape can be drawn onto any rendering device, and the appropriate drawing method will be determined by the dynamic types of *both* the shape and the device. The search can be manually carried out on standard OO languages by means of the tedious *double dispatching* pattern. A more effective alternative is *multi-methods* which are believed to be more expressive, natural and readable than *mono-methods*.

   Multi-methods are found in many new generation OO languages including KEA [99], POLYGLOT [2], CLOS [15], COMMONLOOPS [16], CECIL [26] and DYLAN [120]. The main reason this expressive lingual construct did not find its way into mainstream OO languages is that solving the multiple dispatching problem is (still) believed to be extremely space- and time-intensive, even though the OO research community devoted much effort to find an efficient implementation of multiple dispatching [5, 27, 28, 51, 52, 61, 77, 87, 112].

4. **Object layout** Type conformance forces us to make the layout of a type compatible with that of its supertypes. If a type can have several direct supertypes, then a layout enabling direct field access may not exist. The challenge is in improving the state of the art [53, 68, 101, 113] in layout schemes which optimize both field access time and objects' memory footprint.

The research domain is defined by these four key problems, with the following variations:

1. **Single inheritance vs. Multiple inheritance** In a single inheritance hierarchy each type has at most one direct supertype, which means that the hierarchy takes a tree or forest topology, as in SMALLTALK, OBJECTIVE-C [36], and other OO languages. The hierarchy of Figure 1.1 is an example of a single inheritance hierarchy. Algorithms in the single inheritance setting tend to be more efficient than the general case of multiple inheritance. Some OO languages fall in between these two variations. For example, JAVA has a multiple inheritance *type* hierarchy, but a single inheritance *class* hierarchy.

2. **Incremental- vs. batch- algorithms** We are also interested in *incremental* algorithms where the hierarchy evolves at runtime. The most important kind of change is the addition of types at the bottom of hierarchy, also called *dynamic loading*. (This is the case in JAVA, where types may be added as leaves at run time.) Previous research explored addition of methods to existing nodes, as well as deletion of types and other modifications. We are in a search for *truly incremental* algorithms, i.e., algorithms whose total resource consumption in processing n update operations

is the same as the best *batch* algorithms for processing these n updates submitted together.

3. **Statically- vs. dynamically typed languages** Statically typed languages such as EIFFEL and C++ may provide (partial) type information. The challenge is in utilizing this information at runtime. Conversely, it is often more difficult to find algorithms for dynamically typed languages (sometimes called *dynamic-typing*).

Outline This thesis is organized as follows. Chapter 2 summarizes the contributions of our five papers. These papers are then included as Chapters 3–7, each of which is a self-contained unit and can be read by itself:

1. Chapter 3 describes an efficient algorithm for subtyping test.
2. A space-efficient dispatching technique is presented in Chapter 4.
3. An incremental, constant time, dispatching technique is the subject of Chapter 5.
4. Chapter 6 includes a novel object layout scheme.
5. The penultimate Chapter 7 presents a surprising application of our dispatching techniques in solving the problem of isomorphism of simple types.

Finally, conclusions and directions for future research are given in Chapter 8.

Chapter 2

Contributions

The algorithms which were developed in this thesis are *practical* in the sense that we are not only interested in *theoretical complexity*, but also in how these algorithms handle real data. Therefore, one of our objectives was to collect a *data-set* to evaluate the algorithms for each of the four problems. The data-sets, drawn from ten different programming languages, have now become a de facto standard benchmark.

Forty-four hierarchies were assembled from the following sources:

1. The five multiple inheritance hierarchies (IDL, Laure, JDK 1.1, Ed, Eiffel4) used by Eckel and Gil [53] in their benchmark of object layout techniques.

2. The Flavors hierarchy, representing the *multi-inheritance core* of the Flavors language [98], used by Pugh and Weddell [113, Fig. 5] in their benchmark of object layout techniques.

3. Three newer versions of the JAVA runtime environment (JDK 1.18, JDK 1.22, JDK 1.30).

4. The four hierarchies (Self, Unidraw, LOV, Geode) used in benchmark of row-displacement in multiple inheritance hierarchies [47].

5. The eight hierarchies (Visualworks1, Visualworks2, Digitalk2, Digitalk3, NextStep, ET++, IBM Smalltalk 2, VisualAge 2) used for benchmarking row-displacement and compact-dispatch-tables [132] in single inheritance hierarchies.

6. The ensemble of seven JAVA hierarchies (Corba, HotJava, IBM SF IBM XML, Orbacus, Orbacus Test, Orbix) used in the definition of the "common programming practice" [30], augmented by version 1.3.1 of the Java Development Kit (JDK 1.3.1). Each of these eight hierarchies, was also used both for benchmarking multiple inheritance dispatching algorithms and, after pruning interfaces, for benchmarking single inheritance dispatching algorithms.

7. The two CECIL [26] and DYLAN [120] hierarchies used in all benchmarking of multiple dispatching algorithms [51, 52, 77, 112] contributed by Eric Dujardin.

   We used these hierarchies for benchmarking single-dispatch algorithms, by projecting each multi-method on each of its arguments. (The details are in Section 4.7.)

8. A collection of five other multiple dispatching hierarchies contributed by Wade Holst: Cecil- and Cecil2 are two older versions of the CECIL run time library. Vortex3 is a CECIL compiler written in CECIL, while Vor3 is an old version of this compiler. Harlequin is a commercial implementation of DYLAN including its GUI library.

Table 2.1 shows the number of types n, messages m, and method implementations ℓ, of the 44 hierarchies used in our data-set. The three blocks in the table correspond to single inheritance-, multiple inheritance-, and multiple dispatch- hierarchies. The next column shows that the hierarchies were drawn from ten different OO languages. We see that the hierarchies span a range of sizes, from about a hundred types up to almost 9,000 types. The number of messages m is slightly higher than the number of types, and each message has around five method implementations on average.

We used the following three metrics for evaluating our algorithms: (i) space, (ii) query time, and (iii) the time for creating the encoding. All experiments were run on a Pentium III, 900Mhz, the IBM T22 laptop type 2647-4EG, equipped with 256MB internal memory and running a Windows 2000 operating system.

The publication arising from this thesis are as follows:

Chapter 3 *Efficient Subtyping Tests with PQ-Encoding* [136], in OOPSLA'01 conference and accepted to the TOPLAS journal [70].

In this paper we presented a new subtyping tests scheme named PQ-Encoding (PQE). The median improvement in *space* with the next best algorithm, NHE [90], is by 37%, while the average improvement is 50%. In addition, PQE *subtyping-test time* is constant using only two comparisons, whereas NHE, which is a bit-vector encoding, is non-constant. The *creation time* of PQE is always less than half a millisecond per type.

PQE is called after *PQ-trees*, a data structure previously used in graph theory for finding the orderings that satisfy a collection of constraints. The main reason that our algorithm gives such good results is that PQ-trees use only linear time to process a potentially exponential number of permutation. In a sense, our algorithm chooses an ordering of the types of the hierarchy which satisfies many conflicting constraints. The failing constraints are then collected together and the algorithm tries to find another ordering for as many of these as possible. The process is repeated until all such constraints are satisfied at least in one ordering.

Chapter 4 *Fast Algorithm for Creating Space Efficient Dispatching Tables with Application to Multi-Dispatching* [137], in OOPSLA'02 conference.

In this paper we presented the *type slicing* (TS) technique for dispatching (and subtyping tests). The average improvement in *space* of TS over row-displacement [45, 47] (considered the best for multiple inheritance hierarchies) is by a factor of 4.6, while the median improvement is by a factor of 2.6. In one hierarchy row-displacement uses 1.24MB while TS uses only 40KB! The average improvement in *creation time* is by a factor of 37.4; in the toughest hierarchy the creation time of TS is 4.8 microseconds per method implementation. The cost is in the *dispatching time*, which is

	Language	Hierarchy	n	m	ℓ
Single Inheritance	SMALLTALK	Visualworks1	774	1,170	4,624
	SMALLTALK	Visualworks2	1,956	3,196	13,581
	SMALLTALK	Digitalk2	535	962	3,330
	SMALLTALK	Digitalk3	1,357	2,402	9,444
	SMALLTALK	IBM Smalltalk 2	2,320	4,335	16,288
	SMALLTALK	VisualAge 2	3,241	6,529	26,205
	OBJECTIVE-C	NextStep	311	499	2,115
	C++	ET++	371	296	1,413
	JAVA	SI: JDK 1.3.1	6,681	4,392	23,815
	JAVA	SI: Corba	1,329	222	2,585
	JAVA	SI: HotJava	644	690	2,908
	JAVA	SI: IBM SF	6,626	11,664	88,280
	JAVA	SI: IBM XML	107	131	587
	JAVA	SI: Orbacus	1,053	980	3,821
	JAVA	SI: Orbacus Test	579	368	2,387
	JAVA	SI: Orbix	1,278	535	2,900
Multiple Inheritance	SELF	Self	1,802	2,459	21,753
	C++	Unidraw	614	360	2,331
	EIFFEL	LOV	436	663	2,840
	EIFFEL	Geode	1,318	1,413	9,515
	JAVA	MI: JDK 1.3.1	7,401	5,724	28,683
	JAVA	MI: Corba	1,699	396	3,201
	JAVA	MI: HotJava	736	829	3,397
	JAVA	MI: IBM SF	8,793	14,575	116,152
	JAVA	MI: IBM XML	145	271	945
	JAVA	MI: Orbacus	1,379	1,261	4,996
	JAVA	MI: Orbacus Test	689	379	2,751
	JAVA	MI: Orbix	2,716	786	3,704
	JAVA	JDK 1.18	1,704	-	-
	JAVA	JDK 1.22	4,339	-	-
	JAVA	JDK 1.30	5,438	-	-
	JAVA	JDK 1.1	226	-	-
	FLAVORS	Flavors	67	-	-
	C++	IDL	67	-	-
	LAURE	Laure	295	-	-
	EIFFEL	Eiffel4	1,999	-	-
	EIFFEL	Ed	434	-	-
Multiple Dispatching	DYLAN	Harlequin	666	229	1,016
	DYLAN	Dylan	925	428	1,783
	CECIL	Cecil	932	1,009	4,208
	CECIL	Cecil-	473	592	2,359
	CECIL	Cecil2	472	131	562
	CECIL	Vor3	1,660	328	1,864
	CECIL	Vortex3	1,954	476	2,496
	Total		78,541	70,680	418,839

Table 2.1: Number of types n, messages m, and method implementations ℓ, of the 44 hierarchies used in our data-set

no longer constant, but *logarithmic* in the number of different method implementations. In our data-set, dispatching uses one indirect branch and, on average, only 2.5 binary branches.

TS generalizes an existing dispatching technique for single inheritance hierarchies [60, 100] to also handle multiple inheritance hierarchies. This generalization comes with an increase to the space requirement by a small factor of κ. This factor can be thought of as a metric of the complexity of the topology of the inheritance hierarchy. Surprisingly the same multiple inheritance complexity factor κ is also used in our constant-time dispatching technique (Chapter 5).

We also described an *incremental* subtyping test scheme which is based on theoretical algorithms for range-minima and maintaining order in a list. In the worst case hierarchy, the average time for inserting a new type is as little as 16 microseconds. The *subtyping-test time* is a small constant and the *space requirements* are favorable: always better than the PE scheme [133], and worse than NHE [90] by a median factor of 2.8.

Finally, the paper also has an application to the more general problem of multi-dispatching. The best practical techniques for multiple dispatching known today are *Compressed N-dimensional Tables* (CNT) [5, 52, 87] and *Single-Receiver Projections* (SRP) [77]. Both techniques begin with the same *mono-dispatch stage*, in which c independent single-dispatch queries are executed for a multi-method of arity c. The results of these queries are then used in the *resolution stage* which is technique specific. The mono-dispatch stage in SRP or CNT is currently carried out using either the technique of *selector coloring* [44, 118] or row-displacement [45, 47] for single dispatching. We showed that, in practice, the space requirements of the mono-dispatch stage dominates those of the resolution stage (especially in SRP). Therefore, our TS scheme for single dispatching can significantly reduce the space requirements of multiple dispatching.

Chapter 5 *Incremental algorithms for dispatching in dynamically typed languages* [138], in POPL'03 conference.

Previous theoretical algorithms tend to be impractical due to their implementation complexity and large hidden constant. In contrast, successful practical heuristics, including Vitek and Horspool's *compact dispatch tables* (CT) [132] designed for dynamically typed languages, lack theoretical support. In subjecting CT to theoretical analysis, we are not only able to improve and generalize it, but also provide the first non-trivial bounds on the performance of such a heuristic.

Let n, m, ℓ denote the total number of types, messages, and different method implementations, respectively. Then, the dispatching matrix, whose size is nm, can be compressed by a factor of at most $\iota \equiv (nm)/\ell$. Our main variant to CT achieves a compression factor of $\frac{1}{2}\sqrt{\iota}$. More generally, we describe a sequence of algorithms CT$_1$, CT$_2$, CT$_3$, $\ldots$, where CT$_d$ uses d memory dereferencing operations during dispatch while achieving compression by a factor of (at least) $\frac{1}{d}\iota^{1-1/d}$. This tradeoff between dispatching time and space requirements represents the first bounds on the compression ratio of constant-time dispatching algorithms.

Extending these algorithms to a multiple inheritance setting increases the space by a factor of $(2\kappa)^{1-1/d}$, where κ is the same complexity factor of hierarchies used in our logarithmic dispatching scheme (Chapter 4).

The most important generalization is an *incremental* variant of CT_d for a single inheritance setting. This variant uses at most twice the space of CT_d, and its time of inserting a new type into the hierarchy is optimal. We therefore obtain algorithms for efficient management of dispatching in dynamic-typing, dynamic-loading languages, such as SMALLTALK and even JAVA's `invokeinterface` byte-code instruction.

In our dispatching data-set, the *creation time* of CT_2 is 6.7 micro-seconds per method implementation. The space requirements of the CT schemes are favorable: row-displacement improves, on average, upon CT_2 by 179% (CT_3 by 41%, CT_4 by 12%, and CT_5 by 3%). Even though it seems that row-displacement and CT_5 gives similar compression rates, in one hierarchy CT_5 improves upon row-displacement by a factor of 1,100%, whereas row-displacement improves upon CT_5 by at most 91%. We also note that row-displacement is not suited for dynamic-typing, dynamic-loading languages, and amending row-displacement for dynamic-typing results in doubling its space requirements.

Chapter 6 *Two-Dimensional Bi-Directional Object Layout* [139], in ECOOP'03 conference.

C++ object layout schemes rely on (sometimes numerous) compiler generated fields. In this paper we describe a two-dimensional bi-directional object layout scheme. Our scheme is space optimal, i.e., objects are contiguous, and contain *no compiler generated fields* other than a single type identifier. As in C++ and other multiple inheritance languages such as Cecil and Dylan, our scheme sometimes requires an extra level of indirection to access some of the fields. Using the object-layout data set, we show that our scheme improves field access efficiency over standard implementations, and competes favorably with (the non-space optimal) highly optimized C++ specific implementations. However, our scheme relies on whole-program analysis, which requires only 10 micro-seconds per type, even in very large hierarchies.

After carrying out our research on two-dimensional bi-directional layout, we learnt that similar results were independently obtained by Pugh and Weddell and described in a 1993 technical report [114]. Their work suggests a similar layout algorithm, using fields instead of types, and includes several theoretical bounds on complexity. Our work takes a more empirical slant. We are now in the process of merging these independent research efforts and making a unified journal submission.

Chapter 7 *Efficient algorithms for isomorphisms of simple types* [140], in POPL'03 conference and accepted to a special issue on type isomorphism in the journal "Mathematical Structures in Computer Science" (MSCS) [69].

Surprisingly the techniques developed in our two dispatching papers (Chapters 4–5) are not limited to the OO arena. The *first order isomorphism problem* is to decide whether two non-recursive types using product- and function-type constructors, are isomorphic under the axioms of commutative and associative products, and currying and distributivity of functions over products. We show that this problem can be solved in $O(n \log^2 n)$ time and $O(n)$ space, where n is the input size. This result improves upon the $O(n^2 \log n)$ time and $O(n^2)$ space bounds of the best previous algorithm. We also describe an $O(n)$ time algorithm for the *linear isomorphism problem*, which does not include the distributive axiom, thereby improving upon the $O(n \log n)$ time of the best previous algorithm for this problem.

We now hint on the relationship between type isomorphism and OO implementation. Note first that when substituting function-types for exponentiation and product-types for multiplication, the first order isomorphism problem becomes an instance of the problem of proving an *algebraic equation*, e.g., whether (2.1) holds

$$\left(x^b \cdot \left(y^b \cdot z^d\right)^c\right)^a = (x \cdot y^c)^{a \cdot b} \cdot z^{a \cdot c \cdot d} \,. \tag{2.1}$$

Consider now the left-hand side of (2.1), i.e., the expression $\left(x^b \cdot \left(y^b \cdot z^d\right)^c\right)^a$. Standard algebraic rules show that the term a appears in the exponent of variables x, y, and z. Similarly, term c will appear in the exponent of variables y and z, but not of x. We can therefore describe the effect of each of the terms in the exponent in a tree structure as depicted in Figure 2.1.

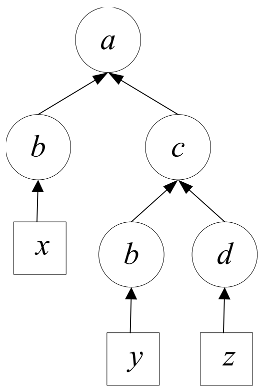


Figure 2.1: The tree structure of the left-hand side of (2.1)

We see in the figure that the leaf x "inherits" the terms b and a, while leaf z "inherits" the terms d, c and a. A similar process of *inheritance* of method implementations occurs in the dispatching problem. Therefore we could successfully borrow dispatching techniques for single inheritance hierarchies such as switching to a dual representation, efficient handling of intervals and segments, and incremental overlaying of segment-partitionings.

Chapter 3

Efficient Subtyping Tests with PQ-Encoding

Chapter Summary
Given a type hierarchy, a subtyping test *is to determine whether one type is a direct or indirect descendant of another type. Such tests are a frequent operation during the execution of object-oriented programs. The implementation challenge is in encoding the hierarchy in a small space, while simultaneously making sure that the tests have efficient implementation. We present a new scheme for encoding multiple and single inheritance hierarchies, which, in the standard benchmark hierarchies, reduces the footprint of all previously published schemes. Our scheme is called* PQ-encoding *(PQE) after* PQ-trees*, a data structure previously used in graph theory for finding the orderings that satisfy a collection of constraints. In particular, we show that in the traditional object layout model, the extra memory requirements for single inheritance hierarchies is zero. In the PQE subtyping tests are constant time, and use only two comparisons. The encoding creation time of PQE also compares favorably with previous results. It is less than a second on all standard benchmarks on a contemporary architecture, while the average time for processing a type is less than one millisecond. Yet, PQE is not an incremental algorithm. Other than PQ-trees, PQE employs several novel optimization techniques. These techniques are applicable also in improving the performance of other, previously published, encoding schemes.*

One of the basic operations in the run time environment of object-oriented (OO) programs is a *subtyping test*. Given an object o and a type b, a subtype test determines whether a, the type of o, is a subtype of b, i.e., a is a direct or indirect descendant of b in the inheritance hierarchy. The subtyping tests (also known as *type inclusion* tests) we refer to are carried out at runtime, and are distinct from static subtyping tests done by the language type checker.

An OO programmer may apply a subtyping test explicitly using dedicated constructs such as JAVA's [7] **`instanceof`**, and SMALLTALK's [71] **`isKindOf:`** method. In addition, several other language constructs are implemented using these tests. For example, subtyping tests are implicit in the execution of type cast operations, e.g., **`?=`** in the EIFFEL programming language [97] and **`dynamic_cast`** in C++ [124]. In JAVA for example, the lack of parametric polymorphism is a common source of such casts. When extracting an element o from a collection class, e.g., **`Vector`**, the type of o is **`Object`**. Therefore, it is typically necessary to cast o to the expected type.

Yet another construct which is implemented using subtyping tests is covariant overriding of arguments in EIFFEL. The compiler is inclined to make subtyping tests in conjunction with calls to methods that use this feature[1].

Also, the covariant nature of array subtyping in JAVA renders subtyping tests necessary in assignments to elements of arrays whose dynamic type is unknown. Consider for example the following code fragment

```
void f(Object x[]) {
   x[1] = new A();
}
```

It may be a bit surprising that the assignment to `x[1]` in `f` requires a subtyping test. To understand why, suppose that function `f` was invoked with a value of type `B[]` (an array of elements of type `B`) as an actual argument `x`, e.g.,

```
f(new B[3]);
```

This invocation is correct since type `B[]` is a subtype of `Object[]` (an array of objects). However, the assignment

```
x[1] = new A();
```

is legal only if type `A` is a subtype of type `B`. Otherwise, the runtime environment raises an `ArrayStoreException` exception.

Finally, we note that subtyping tests may also be part of the implementation of exception handling in JAVA, C++, and other languages. The following code excerpt is an example of a **`try`** block, followed by a **`catch`** clause and block.

```
try { f(); ...}
catch(B b) {...}
```

Suppose that an object o is **`throw`**n from the **`try`** block, e.g., from within function `f()`. Then, the program should execute the **`catch`** block, but only if type `B` is a supertype of the thrown object's type. Thus, the **`catch`** clause can be implemented with a subtyping test. In JAVA, the subtyping test is with respect to the dynamic type of o, while in C++ it is with respect to the static type of o. However, in both cases, this test must be carried out at run-time.

Outline The remainder of this chapter is organized as follows. The subtyping test problem is defined in Section 3.1. Some straightforward solutions for this problem are described in Section 3.2. Section 3.3 makes some pertinent definitions. The data set of the 13 hierarchies used in our benchmarking is presented in Section 3.4. A survey of prior research is the subject of Section 3.5. This section also describes the *slicing* technique of partitioning a hierarchy for the purpose of subtyping tests. The technique is common to

[1]Various mechanisms of static analysis have been proposed to eliminate this requirement, but none of these have been implemented.

many previous algorithms for the problem; it also stands as the basis of the PQE algorithm which is described in Section 3.6. Section 3.7 presents our new optimization techniques, improving instruction count, test time and encoding length. Section 3.8 presents the results of running these algorithms on our benchmark. Finally, some open problems and directions for future research are mentioned in Section 3.9. Section 3.10 demonstrates the inner workings of the PQ-tree data structure.

3.1 Problem Definition

The problem we deal with is defined formally as follows. A *hierarchy* is a partially ordered set $(\mathcal{T},\preceq)$ where $\mathcal{T}$ is a set of types and $\preceq$ is a reflexive, transitive and anti-symmetric *subtype relation*. If a,b are types, and $a \preceq b$ holds, we say that they are *comparable*, that a is a *subtype* of b and that b is a *supertype* of a. Given a hierarchy $(\mathcal{T}, \preceq)$, $|\mathcal{T}| = n$, the *subtyping problem* is to build a data structure supporting queries of the sort $a \preceq b$. This data structure is called an *encoding* of the hierarchy.

The subtyping problem has enjoyed considerable attention recently (see e.g., [1, 22, 25, 34, 58, 59, 62, 74, 75, 84, 90, 110, 115, 127, 133]), the challenge being in simultaneously optimizing its four complexity measures:

1. *Space.* Encoding methods associate certain data with each type. We measure the average number of bits per type, also called the *encoding length*.

   Note that we do not include in the measure the space consumed by each object. Although the space overhead per object depends on the object layout model, it is usually assumed that each object includes a pointer to a type information record. Optimizing object space is beyond the scope of this thesis.

2. *Instruction count.* This is the number of machine instructions in the *test code*, on a certain hardware architecture. There are indications [133] that the space consumed by the test code, which can appear many times in a program, can dominate the encoding length. An encoding is said to be *uniform*[2] if there exists an implementation of the test code in which the instruction count does not depend on the size of the hierarchy. Only uniform encodings will interest us.

3. *Test time.* The time complexity of the test code is of major interest. Since the test code might contain loops, the time complexity may not be constant even in uniform encodings. Our main concern here are constant time encodings (which are always uniform). To improve timing performance, loops of non-constant time encodings may be unrolled, giving rise to non-constant instruction count, without violating the uniformity condition. (Bit-vector encoding, presented in Section 3.5, is an example of a uniform encoding which is non-constant time.)

   In explicit subtyping tests and in type casts, the test $a \preceq b$ is not entirely arbitrarily in the sense that the supertype b is known at compile time. The test code can then

[2]The term is borrowed from circuit complexity. A family of circuits for the size dependent incarnations of a certain problem is called uniform, if this family can be generated by a single Turing machine.

be *specialized*, by precomputing values which depend only on b and emitting them as part of the test code. Specialization thus benefits both instruction count and test time, and may even reduce the encoding length.

4. *Encoding creation time.* Another important complexity measure is the time for generating the actual encoding, which can be large.

   It is essential that a compiler will be able to finish its computation in a reasonable time. In many cases, this is not possible. For example, the problem of finding an optimal bit-vector encoding was proved to be intractable [75]. Heuristics of bit vector encoding [25, 62, 74, 84, 90, 115] offer a tradeoff between creation time and encoding length.

   Most of the previous work assumed, as we shall do here, that the entire type hierarchy is supplied at compile time. JAVA, E [76] and many other languages allow types to be dynamically loaded at runtime. If the encoding creation time is sufficiently small, then the encoding can be *recomputed* whenever such a load occurs. An active research topic is to find truly incremental algorithms, which can quickly *update* the encoding.

3.2 Straightforward Solutions

The most obvious (uniform) representation as a *binary matrix* gives constant subtyping tests, but the encoding length is n. This method is useful for small hierarchies and is used e.g., for encoding the JAVA *interfaces* hierarchy [88] in CACAO 64-bit JIT compiler [72, 89]. The quadratic space becomes very noticeable in large hierarchies. For example, one of the hierarchies in our data set has 5500 types which give rise to 3.8MB binary matrix. The binary matrix encoding can be (non-uniformly) implemented using a zero encoding length and $O(n)$ instruction count: relying on specialization, the test code for $a \preceq b$ then checks whether a is among the possibly $O(n)$ descendants of b. More generally, a non-uniform encoding is tantamount to representing the encoding data structure as part of the test code, and therefore will not interest us.

The observation that stands behind the work on subtyping tests is that the binary matrix representation is in practice very sparse, and therefore susceptible to massive optimization. Nevertheless, the number of partially ordered sets (*posets*) with n elements is $2^{\Theta(n^2)}$, so the representation of some posets requires $\Omega(n^2)$ bits[3]. Thus, the encoding length is $\Omega(n)$. In other words, for arbitrary hierarchies the performance of the binary matrix implementation is asymptotically optimal.

Let the relation $\prec_{\mathrm{d}}$ be the *transitive reduction* of $\preceq$, i.e., a minimal relation whose *transitive closure* is $\preceq$. More precisely, relation $\prec_{\mathrm{d}}$ is defined by the condition that $a \prec_{\mathrm{d}} b$ if and only if $a \preceq b, a \neq b$, and there is no $c \in \mathcal{T}$ such that $a \preceq c \preceq b, a \neq c \neq b$.

Figure 3.1 depicts a directed acyclic graph (DAG) representation of a hierarchy which will serve as the running example of this chapter.

[3]The number of bipartite graphs with n elements is clearly $2^{\Theta(n^2)}$, and every bipartite graph is also a poset.

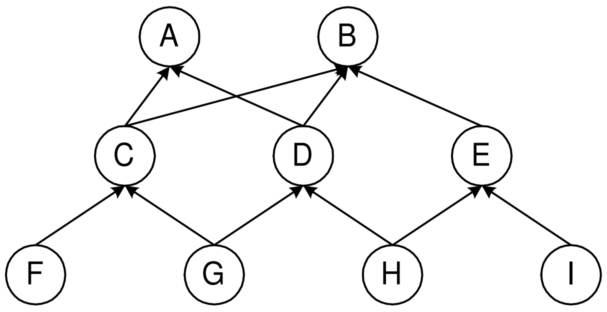


Figure 3.1: A small example of a multiple subtyping hierarchy

We employ the usual convention that edges are directed from the subtype to the supertype, and that types drawn higher in the diagram are considered greater in the subtype relationship. Thus, the figure specifies (for example) that $\mathsf{G} \prec_{\mathrm{d}} \mathsf{C}$ and $\mathsf{H} \preceq \mathsf{A}$. In total in this hierarchy, $n = 9$, $|\prec_{\mathrm{d}}| = 11$, and $|\preceq| = 27$.

Another obvious solution to the subtyping problem is *DAG-encoding*, which is based on the DAG defined by types as nodes and edges from $\prec_{\mathrm{d}}$. In this encoding, a list of parents is stored with each type, resulting in total space of $|\prec_{\mathrm{d}}|\lceil\log n\rceil$ bits[4] in an idealized bit-efficient representation. The DAG encoding length is therefore

$$\frac{|\prec_{\mathrm{d}}|}{n} \cdot \lceil\log n\rceil.$$

The average number of parents, $|\prec_{\mathrm{d}}|/n$, tends to be small; We will see that it is less than 2 in all the standard benchmark hierarchies. Unfortunately, a subtyping test in DAG-encoding is $O(n)$ time.

The *Closure-encoding* presents another obvious tradeoff between space and test time. In this encoding, with each type we store the list of *all* of its ancestors using a simple sorted array representation. A subtyping test is then implemented using a binary search in $O(\log n)$ time. Since each entry in this array requires $\lceil\log n\rceil$ bits, the encoding length is

$$\frac{|\preceq|}{n} \cdot \lceil\log n\rceil.$$

Theoretically superior representations of this list include *Q-fast tries* [134], which achieve deterministic $O(\sqrt{\log n})$ time, or the randomized stratified trees (also called *van Emde Boas data structure*) [128, 129], which achieve $O(\log\log n)$ time. Another alternative are perfect hash tables [64] which give $O(1)$ lookup time. In moderately sized tables we expect the simple binary search algorithm to outperform the asymptotically better competitors. Also, these sophisticated data structures come at a cost of an increase of the encoding length by factors which can be prohibitively large.

The binary matrix, DAG-, and Closure-encoding are not very appealing techniques. Previous contributions in this field included many sophisticated encoding schemes which come close to DAG-encoding in space, while keeping the test time constant or “almost” constant.

[4]Here and henceforth, all logarithms are based two.

An important special case of the problem is single inheritance, which occurs when the hierarchy DAG takes a tree or forest topology as mandated by the rules of languages such as SMALLTALK [71] and OBJECTIVE-C [36]. The general case of multiple inheritance is more difficult, and will be our main concern here.

3.3 Definitions

Given a type $a \in \mathcal{T}$, we define the following sets: $\mathrm{descendants}(a)$ and $\mathrm{ancestors}(a)$ (the set of subtypes and supertypes of a, respectively), as well as $\mathrm{children}(a)$ and $\mathrm{parents}(a)$ (the set of *immediate* subtypes and supertypes of a, respectively). More precisely,

$$\begin{aligned}\mathrm{descendants}(a) &\equiv \{b \in \mathcal{T} \mid b \preceq a\}\\ \mathrm{ancestors}(a) &\equiv \{b \in \mathcal{T} \mid a \preceq b\}\\ \mathrm{children}(a) &\equiv \{b \in \mathcal{T} \mid b \prec_{\mathrm{d}} a\}\\ \mathrm{parents}(a) &\equiv \{b \in \mathcal{T} \mid a \prec_{\mathrm{d}} b\}\end{aligned} \tag{3.1}$$

Also for $a \in \mathcal{T}$, the value $\mathrm{level}(a)$ is the length in nodes of the longest directed (upgoing) path starting from a. The *height* of the hierarchy is the maximal level among all types in $\mathcal{T}$. The k^{th}*-level* of the hierarchy is the set of all types a for which $\mathrm{level}(a) = k$.

$$\begin{aligned}\mathrm{level}(a) &\equiv 1 + \max\{\mathrm{level}(b) \mid b \in \mathrm{parents}(a)\}\\ \mathrm{height}(\mathcal{T}) &\equiv \max\{\mathrm{level}(a) \mid a \in \mathcal{T}\}\end{aligned} \tag{3.2}$$

(In the above definition of $\mathrm{level}(a)$, the maximum over an empty set is defined as zero. In other words, nodes without any parents are defined as being in level 1.)

In Figure 3.1 we have that

$$\begin{aligned}\mathrm{descendants}(\mathsf{A}) &= \{\mathsf{A}, \mathsf{C}, \mathsf{D}, \mathsf{F}, \mathsf{G}, \mathsf{H}\}\\ \mathrm{ancestors}(\mathsf{F}) &= \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{F}\}\\ \mathrm{children}(\mathsf{A}) &= \{\mathsf{C}, \mathsf{D}\}\\ \mathrm{parents}(\mathsf{F}) &= \{\mathsf{C}\}\\ \mathrm{level}(\mathsf{F}) &= 3\end{aligned}$$

This hierarchy has three levels: with two, three, and four types, respectively.

The following definitions will also become pertinent:

$$\begin{aligned}\mathrm{roots}(\mathcal{T}) &\equiv \{a \in \mathcal{T} \mid \mathrm{parents}(a) = \emptyset\}\\ \mathrm{leaves}(\mathcal{T}) &\equiv \{a \in \mathcal{T} \mid \mathrm{children}(a) = \emptyset\}\end{aligned} \tag{3.3}$$

In Figure 3.1 we have that

$$\begin{aligned}\mathrm{roots}(\mathcal{T}) &= \{\mathsf{A}, \mathsf{B}\}\\ \mathrm{leaves}(\mathcal{T}) &= \{\mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}\}.\end{aligned}$$

3.4 Data Set

To benchmark the algorithms, we started from the 9 multiple inheritance hierarchies used by Eckel and Gil [53] in their benchmark of object layout techniques. Three new JAVA hierarchies (newer versions of the JAVA runtime environment), as well as the Cecil compiler hierarchy [26] were added to this benchmark. In total, our data set represents an array of large hierarchies drawn from six different OO languages. In particular, the set includes all multiple inheritance hierarchies used in previous studies of encoding schemes [74, 90, 133]. Eckel and Gil [53] gave a detailed description of these hierarchies. One of their findings is that many topological properties of typical hierarchies are similar to those of balanced trees. This makes it possible to find more efficient encodings for hierarchies used in practice. Comparison of different encoding schemes is done over these 13 hierarchies which have now become a de facto standard benchmark.

The hierarchies in the data set are enumerated in ascending order of size in Table 3.1. We see that the number of types ranges between 66 and 5,438. In total the 13 hierarchies represent over 19,500 types.

Hierarchy	n	$\lvert\prec_{\mathrm{d}}\rvert/n$	$\lvert\preceq\rvert/n$	α[a]	β[b]	γ[c]	$\lvert\mathcal{T}'\rvert/n$
IDL	66	0.98	3.83	8	6	7	15%
JDK 1.1	225	1.04	3.17	7	6	8	15%
Laure	295	1.07	8.13	16	11	9	18%
Ed	434	1.66	7.99	23	10	9	61%
LOV	436	1.71	8.50	24	9	9	62%
Unidraw	613	0.78	3.02	9	8	10	4%
Cecil	932	1.21	6.47	23	12	10	33%
Geode	1,318	1.89	13.99	50	13	11	75%
JDK 1.18	1,704	1.10	4.35	16	9	11	18%
Self	1,801	1.02	29.89	40	16	11	9%
Eiffel4	1,999	1.28	8.78	39	17	11	46%
JDK 1.22	4,339	1.19	4.37	17	9	13	22%
JDK 1.30	5,438	1.17	4.37	19	9	13	21%

[a] $\max\{|\mathrm{ancestors}(a)| \mid a \in \mathcal{T}\}$
[b] height
[c] $\lceil \log n \rceil$

Table 3.1: Topological properties of hierarchies in the data set

Table 3.1 gives also some of the topological properties of the hierarchies. Examining the third column in the table we see that the average number of parents, $|\prec_{\mathrm{d}}|/n$, is always less than 2. On the other hand, the average number of ancestors, $|\preceq|/n$, can be large. In the Self hierarchy a type has in average almost 30 ancestors! The maximal number of ancestors plays an important factor in the complexity of some of the algorithms. We see that there exists a type in the Geode hierarchy which has 50 ancestors in total. In comparing the height of the hierarchy with $\log n$ we see that the hierarchies are shallow; their height is similar to that of a balanced binary tree.

We can learn a bit more on the topology of inheritance hierarchy by considering the set $\mathcal{T}'$, which can be thought as the *multiple inheritance core* of the hierarchy. Formally, a type is in the core if it has a descendant with more than one parent. Conversely, the set $\mathcal{T} \setminus \mathcal{T}'$ is a collection of maximal subtrees discovered in a bottom-up traversal of the hierarchy. It was previously noticed [133] that encoding is easier if the core is considered first, and the *bottom trees* of $\mathcal{T} \setminus \mathcal{T}'$ are added to the encoding later. In Table 3.1 we see that in most hierarchies the core is rather small, typically less than half the number of types. Treating the core and the bottom trees separately will reduce the run time of our encoding algorithm.

3.5 Previous Work

This section gives an overview of various encoding methods proposed in the literature. We describe the data structure used in each such encoding, and how it is deciphered to implement subtyping tests. Little if any attention is devoted to describing the actual generation of the data structure and the theory behind it.

3.5.1 Encoding of Single Inheritance Hierarchies

Relative numbering

Perhaps the most elegant encoding algorithm is *relative numbering* [119] (also called *Schubert's numbering*) which guarantees both optimal encoding length of $\lceil \log n \rceil$ bits and constant time subtyping tests. However, these achievements are only possible in a single inheritance hierarchy. For a type $b \in \mathcal{T}$, let r_b denote its ordinal (i.e., an integer in the range $1, \ldots, n$) in a postorder traversal of $\mathcal{T}$. A basic property of postorder traversal is that

$$r_b = \max\{r_a \mid a \preceq b\}. \tag{3.4}$$

Let l_b be defined by

$$l_b = \min\{r_a \mid a \preceq b\}. \tag{3.5}$$

Combining (3.4) and (3.5) with the fact that in postorder traversal the descendants of any type are assigned consecutively, we find that $a \preceq b$ iff

$$l_b \leq r_a \leq r_b. \tag{3.6}$$

Thus, in relative numbering, each type a is encoded by an interval $[l_a, r_a]$ as exemplified by Figure 3.2.

In the figure we have (for example) that $\mathrm{descendants}(\mathsf{D}) = \{\mathsf{D}, \mathsf{H}, \mathsf{I}\}$. The r-descriptor of each of these descendants, i.e., $r_{\mathsf{D}} = 8$, $r_{\mathsf{H}} = 6$ and $r_{\mathsf{I}} = 7$, fall within the interval $[6, 8]$ associated with type D. No other r-descriptor falls in this interval.

Recall that since b is known at compile time, values which depend only on b can be precomputed. Henceforth, such values are marked by a "#" prefix. With this notation, we

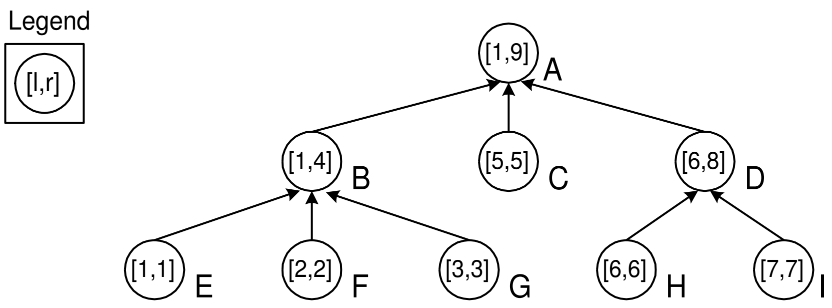


Figure 3.2: Relative numbering in a tree hierarchy

write (3.6) as

$$\#l_b \leq r_a \leq \#r_b. \tag{3.7}$$

We note that $\#l_b$ and $\#r_b$ are compile-time constants and hence the test (3.7) can be specialized by eliminating the memory fetch of these. In doing so, we find that l_b is not part of the encoding, bringing down the encoding length of relative numbering to $\lceil \log n \rceil$.

Relative numbering is used in CACAO [72,89] to represent the JAVA class inheritance hierarchy [88] (Recall that the binary matrix is used in CACAO for the interface hierarchy.) Range-compression [1], described below, is a generalization of relative numbering for multiple inheritance.

Cohen's encoding

A variant of Dijkstra's displays [43] is Cohen's encoding [29]. His encoding is yet another example of an algorithm initially designed for single inheritance, and later generalized to multiple inheritance. (The generalized algorithm, known as Packed Encoding [133], will be described below.) Cohen's encoding relies on hierarchies being relatively shallow, and more so, on types having a small number of ancestors. As Table 3.1 shows, this is indeed the case in some of our multiple inheritance hierarchies. A type a is associated an array r_a of size

$$\mathrm{level}(a) \leq |\mathrm{ancestors}(a)|$$

(in single inheritance, $\mathrm{level}(a) = |\mathrm{ancestors}(a)|$), with entries for each

$$b \in \mathrm{ancestors}(a).$$

Specifically, each type b, $b \succeq a$, is stored in location $\mathrm{level}(b)$ in array r_a. Thus, the test whether $b \succeq a$ is carried out by checking whether b indeed occurs in location $\mathrm{level}(b)$ of array r_a. The encoding is optimized by storing not b itself in this location, but rather an *id*, which is unique among all types in its level.

Since different levels come in different sizes, some id's may require fewer bits than others. Typically, an id is stored in either a single byte or in a 16 bits word. It is even possible to pack several id's into a single byte. As a result of this compression the entries of r_a, which are not of equal size, cannot be referenced using ordinary array access operations.

We say that r is a *pseudo-array,* and use the notation $r@\ i$ instead of $r[i]$ for denoting pseudo-array access. Pseudo-arrays are only used if the index i is always known at compile time. Therefore, a pseudo-array access is the same as record member selection, and is no slower than a non-pseudo array access. (If several pseudo-array entries are packed together in a single byte, then the required shift and mask operations may slow down this operation in comparison to normal array accesses.)

Cohen's encoding stores with each type a its level, $l_a = \mathrm{level}(a)$, its unique id within this level id_a, as well as the pseudo-array r_a, such that for each $b \in \mathrm{ancestors}(a)$,

$$r_a @\ l_b = \#\,\mathrm{id}_b\,. \tag{3.8}$$

The test $a \preceq b$ is carried out by checking that $l_a \geq \#l_b$ and then that (3.8) holds. Note that l_b is known at compile time.

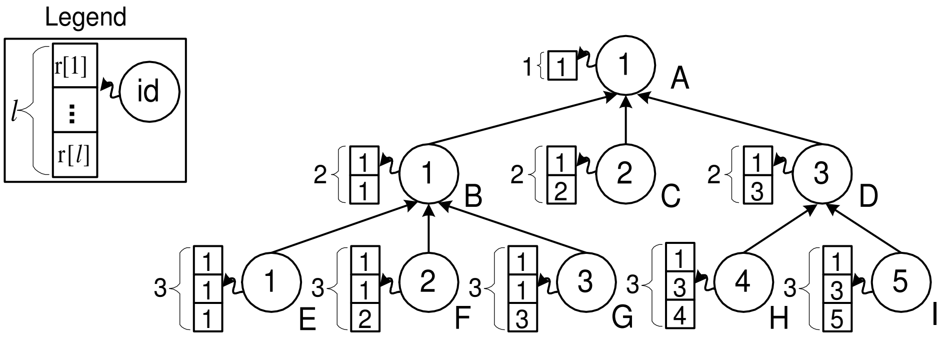


Figure 3.3: Cohen's encoding of the tree hierarchy of Figure 3.2

Consider for example the 3-level tree hierarchy depicted in Figure 3.3. As shown in the figure, each type has a (pseudo-) array with at most 3-identifiers. The pseudo array of type H in the 3*rd* level has three entries: $r_{\mathsf{H}}@\ 1 = 1$ since the ancestor of H at the 1*st* level is A, and $\mathrm{id}_{\mathsf{A}} = 1$. Similarly, the ancestor at the 2*nd* level is D, $\mathrm{id}_{\mathsf{D}} = 3$. The last entry of this array stores the id of H.

Also observe that in the figure how id's are reused at different levels. For example, for types C and F which are at different levels we have that $\mathrm{id}_{\mathsf{C}} = \mathrm{id}_{\mathsf{F}} = 2$.

The array boundary check $l_a \geq \#l_b$ in Cohen's encoding is inelegant. We observe that it can be eliminated at the price of allocating globally unique id's. Then, it is possible to concatenate the arrays, making sure that the largest array is at the end. Even if there is an overflow in the array access $r_a[l_b]$, the location found will not contain id_b.

Jalapeño [4], the IBM implementation of the JAVA virtual machine (JVM), uses Cohen's algorithm for subtyping tests where the supertype is a class. The main reason is that this encoding is incremental, whereas vanilla relative numbering is not.

3.5.2 Encodings of Multiple Inheritance Hierarchies

Packed encoding

Cohen's algorithm was generalized to the multiple inheritance setting by Vitek et al. [133] into what is called *Packed Encoding* (PE) and *Bit-Packed Encoding* (BPE), which are both constant-time methods. Cohen's algorithm, PE, BPE and our algorithm share a common theme: *slicing*, in which the set $\mathcal{T}$ is partitioned into disjoint *slices* (sometimes called buckets) $S_1, \ldots, S_k$. For each slice S_i we store the entire information required to answer queries of the sort $a \preceq b$, $a \in \mathcal{T}$ and $b \in S_i$, i.e., queries in which the supertype is drawn from S_i. Type a has a pseudo-array r_a of length k, where $r_a @\ i$ holds information for slice S_i. In essence, we store, in a very compressed format, the set of descendants of each element in S_i. The compression is possible since there is a great deal of sharing in the descendants set of different members of S_i.

PE associates with each type $a \in \mathcal{T}$ a unique integer id_a within its slice s_a, so that a is identified by the pair $\langle s_a, \mathrm{id}_a \rangle$. Also associated with a is a byte array r_a, such that for all $b \in \mathrm{ancestors}(a)$, index s_b stores id_b, i.e.,

$$r_a[s_b] = \#\, \mathrm{id}_b\,. \tag{3.9}$$

A necessary and sufficient condition for $a \preceq b$ to hold is then (3.9). It should be clear that no two ancestors of a can be on the same slice. Thus, the number of slices is at least the size of the largest set of ancestors. Checking the fifth column of Table 3.1 we see that some hierarchies require 40 slices or more.

Comparing (3.9) with (3.8), we see that slices play a role similar to that of levels in Cohen's algorithm. In fact, Cohen's algorithm partitions the hierarchy into $\mathrm{height}(\mathcal{T})$ anti-chains[5], while PE partitions the hierarchy into anti-chains where no two elements in an anti-chain have a common descendant. Fall [58], who observed that this technique might be used for subtyping tests, noted that it is NP-hard to find a minimal such partition, and stopped short of finding a constant time subtyping test. The heuristic suggested by Vitek et al. [133] along with the constant time subtype test made PE viable. Based on this heuristic, Palacz and Vitek [110] recently gave an incremental implementation of PE.

Vitek, Horspool and Krall's PE algorithm constrains each slice to a maximum of 255 types, so that id_a can always be represented by a single byte. The encoding length is then $8k$, where k is the number of slices. The inventors of PE observed that k is usually the maximal number of ancestors unless multiple inheritance is heavily used. Thus, even though the general problem is intractable, their heuristics often finds an optimal solution.

Consider Figure 3.4 for an example of PE representation of the hierarchy of Figure 3.1. The types of the hierarchy are partitioned into five different slices: $S_1 = \{\mathsf{A}\}$, $S_2 = \{\mathsf{B}\}$, $S_3 = \{\mathsf{D}\}$, $S_4 = \{\mathsf{C}, \mathsf{E}\}$, and $S_5 = \{\mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}\}$. This is the smallest possible number of slices, since type G (for example) has five ancestors.

The only difference between BPE and PE is that BPE permits two slices or more to be represented within a single byte. Thus, in BPE r_a is a pseudo-array, and the array access

[5] An *anti-chain* is a set of types where no two types are comparable. Clearly, each level is an anti-chain.

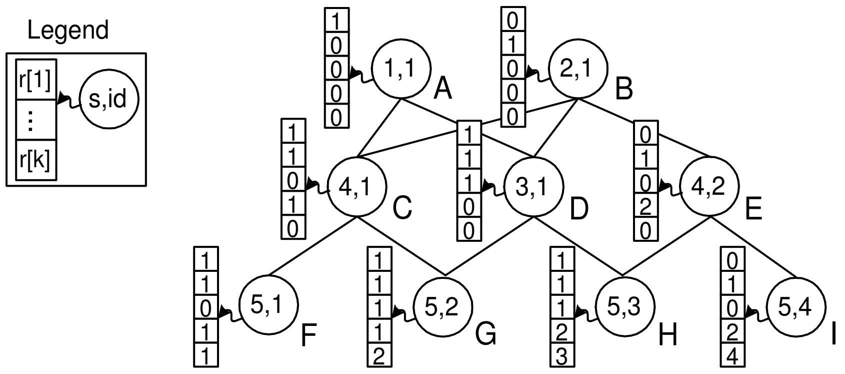


Figure 3.4: PE representation of the hierarchy of Figure 3.1

in (3.9) becomes a pseudo-array access:

$$r_a@\ s_b = \#\,\mathrm{id}_b\,. \tag{3.10}$$

Starting from Figure 3.4 we can represent slices S_1, S_2 and S_3 using a single bit, S_4 using two bits, and S_5 in three bits, for a total of seven bits, which can fit into a single byte.

Bit-vector encoding

One of the most explored directions in prior art is *bit-vector encoding* [25, 62, 74, 75, 84, 90, 115]. In this scheme, each type a is encoded as a vector vec_a of β bits. If $\mathrm{vec}_a[i] = 1$ then we say that a has *gene* i. Let $\phi(a)$ be the set of genes of a. Relation $a \preceq b$ holds iff $\phi(a) \supseteq \phi(b)$, which can be easily checked by masking vec_b against vec_a, specifically, applying the test:

$$\mathrm{vec}_b \ \ \textsf{and} \ \ \mathrm{vec}_a = \mathrm{vec}_b\,. \tag{3.11}$$

Figure 3.5 gives an example of a bit-vector encoding of the hierarchy of Figure 3.1 which uses 6 genes.

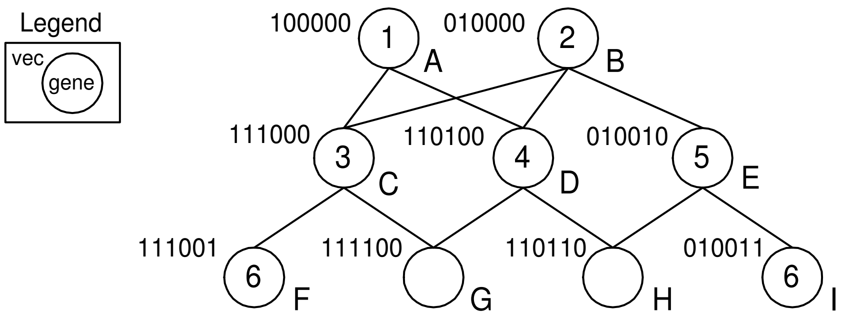


Figure 3.5: Bit-vector encoding of the hierarchy of Figure 3.1. (We only write the genes a type adds to its parents.)

The set of genes of type D (for example) is $\phi(\mathsf{D}) = \{1, 2, 4\}$, and thus $\mathrm{vec}_{\mathsf{D}} = 110100$. The genes of the ancestors of type D are contained in $\phi(\mathsf{D})$, and every other type has at least one gene not in $\phi(\mathsf{D})$.

Bit-vector encoding effectively embeds the hierarchy in the lattice of subsets of the set $\{1, \ldots, \beta\}$. It is always possible to do so by setting $\beta = n$ and in letting vec_a be the row of the binary matrix which corresponds to a. A simple counting argument shows that β must depend on the size of the hierarchy. Hence, bit-vector encoding is non-constant time, but it is uniform. For efficiency reasons, the implicit loop in (3.11) can be unrolled, giving rise to a non-constant instruction count.

The challenge is in finding the minimal β for which such an embedding of the hierarchy in a lattice is possible. Although the problem is NP-hard [75], several good heuristics were proposed, including Kaci et al. [84] work, Caseau's *Compact Hierarchical Encoding* [25], later improved by Habib et al. [74]. Currently, *Near Optimal Hierarchical Encoding* (NHE), due to Krall et al. [90], is the best general bit vector encoding. Better results can be obtained for the special case of single inheritance by *dichotomic encoding* [115] and its *polychotomic encoding* generalization [62].

Range compression

It is only natural to ask then whether it is possible to promise *constant encoding length*, while maintaining uniformity and "almost constant" time. An affirmative answer to this question was given by Agrawal et al. [1] in their *range-compression* encoding which generalizes relative numbering. Range compression encodes each type b as an integer id_b, with its ordinal in a postorder scan of a certain spanning forest of the hierarchy. Then, the set $\Phi(b)$ of id's of the descendants of b,

$$\Phi(b) = \{\mathrm{id}_a \mid a \in \mathrm{descendants}(b)\}, \tag{3.12}$$

is represented by an array of consecutive disjoint intervals

$$[l_b@\ 1, r_b@\ 1], [l_b@\ 2, r_b@\ 2], \ldots, [l_b@\ \gamma(b), r_b@\ \gamma(b)].$$

Thus, $a \preceq b$ iff

$$\#l_b@\ i \leq \mathrm{id}_a \leq \#r_b@\ i \tag{3.13}$$

holds for *some* i, $1 \leq i \leq \gamma(b)$. In single inheritance, all descendants of a type are assigned consecutive numbering in a postorder traversal, and therefore the set (3.12) can be represented using a single interval. The encoding then degenerates to relative numbering.

Figure 3.6 gives a range-compression encoding of the hierarchy of Figure 3.1. We have for example

$$\Phi(\mathsf{B}) = \{1, 2, 3, 5, 6, 7, 8, 9\},$$

which can be represented as two intervals $[1, 3]$ and $[5, 9]$. Thus, $l_{\mathsf{B}} = \langle 1, 5\rangle, r_{\mathsf{B}} = \langle 3, 9\rangle$ and $\gamma(\mathsf{B}) = 2$.

Examining (3.13) we see that only id_a has to be stored for a type a, since everything else is specialized into the subtyping test site. The specialization reduces the encoding length to $\lceil \log n \rceil$, but at a price of increasing the instruction count from constant to $\gamma(b)$, which can be in the order of n. In all of our hierarchies however, the average of $\gamma(b)$

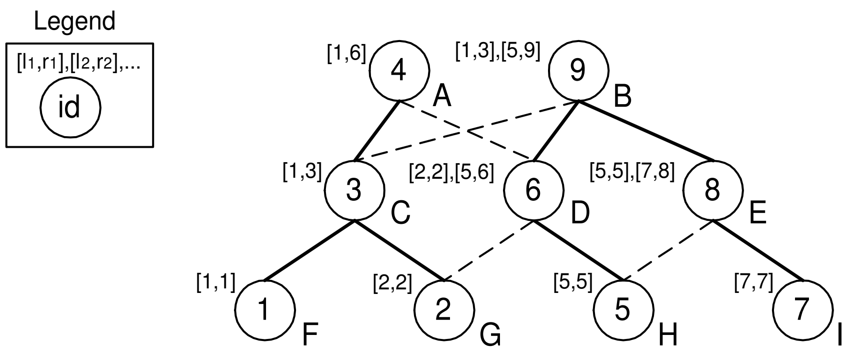


Figure 3.6: Range-compression encoding of the hierarchy of Figure 3.1. (Edges of the spanning forest are in bold.)

over all $b \in \mathcal{T}$ was always less than 2. The maximal $\gamma(b) = 55$ was found in the Geode hierarchy.

The usual straightforward implementation of range compression requires $O(\gamma(b))$ time. If $\gamma(b)$ is large then a binary search on (3.13) reduces the time to $O(\log \gamma(b))$. Note that this faster implementation does nothing to improve the instruction count in the specialized implementation which remains $\Omega(\gamma(b))$.

Other non-constant encoding techniques were used in large data- and knowledge-bases, e.g., modulation techniques [58,84], sparse terms encoding [59], and representation using union of interval orders [22]. The common objective is a small average, rather than worst-case, time for testing, which may be considered unsuitable for an implementation of the runtime environment of OO languages.

3.6 PQ-Encoding

This section describes PQ-encoding (PQE), our new encoding scheme, which achieves the smallest space requirements among all previously published encodings. In a nut-shell, PQE combines the ideas of *relative numbering* with *slicing* as used in PE and BPE.

The essence of relative numbering is in the (global) *consecutiveness property*, i.e., the requirement that the descendants of any given type are numbered consecutively; this property makes it possible to represent the entire set of descendants as a pair of two integers: the end points of the interval. In single inheritance, the consecutiveness property is satisfied by the numbering of a simple postorder visit. For multiple inheritance hierarchies, it is only natural to try to generalize relative-numbering by replacing the postorder visit by a DFS of the inheritance graph. Two issues must be addressed in order to make such a generalization work.

1. The encoding must chose one DFS visit of the inheritance hierarchy from many different such visits, which may lead to *essentially* different orderings of the nodes. (Note that different DFS visits of a single inheritance hierarchy give rise to essentially the same relative numbering encoding.)

2. In general, it is not guaranteed that there exists any single numbering which satisfies the consecutiveness property. Therefore, the generalization must handle hierarchies in which the consecutiveness property cannot be satisfied.

As explained in the previous section, the range-compression technique of Agrawal et al. [1] addresses these issues by applying a heuristic for choosing a DFS. Also, if this heuristic fails, i.e., in case the set of descendants of a certain type does not fill up a single range, then this set is represented as a collection of ranges.

PQE uses two techniques in the generalization of relative numbering:

1. Employing a sophisticated algorithmic tool, namely *PQ-trees*, for efficiently considering together even an exponential number of orderings. In particular, if there exists *any ordering* of the hierarchy which satisfies the global consecutiveness property, then the PQ-trees technique is guaranteed to find one in $O(|\preceq|)$ time.

2. Using the slicing technique to make sure that subtyping tests require constant time, even if no ordering which satisfies the consecutiveness property exists.

We first (Section 3.6.1) explain data structure used by the encoding and the implementation of constant time subtyping tests. Section 3.6.2 explains the slicing technique in greater detail. In Section 3.6.3 we describe the PQ-trees data structure. Section 3.6.4 shows how it is used to find a PQ-encoding.

3.6.1 Subtyping Tests in the PQ-Encoding

The set of types is partitioned into disjoint *slices*, and each type has a distinct id with respect to *each* of the slices. Specifically, let k denote the number of slices. Then, for each type three pieces of data are stored:

1. an integer s_a, $1 \leq s_a \leq k$, which is the number of the slice to which a belongs,

2. a pseudo-array id_a of length k, such that $\mathrm{id}_a @\ i$ is the id of type a with respect to slice i, $1 \leq \mathrm{id}_a @\ i \leq n$ and

3. an interval $[l_a, r_a]$, represented as a pair of integers, $1 \leq l_a \leq r_a \leq n$, which are the smallest and the largest id (with respect to slice s_a) of the descendants of a.

In total $k + 3$ integers are stored for each type. Our main effort, for which we will harness PQ-trees, is to minimize the number of slices k. Fine tuning of the representation as discussed below in Section 3.7 may make the encoding length less than $(k + 3)\lceil \log n \rceil$.

The selection of id's and intervals is such that subtyping tests can be made using two comparisons. Specifically, $a \preceq b$ iff

$$\#l_b \leq \mathrm{id}_a @\ s_b \leq \#r_b. \tag{3.14}$$

Thus, subtyping tests begins with the pseudo-array access $\mathrm{id}_a @\ s_b$ which finds the id of a with respect to the slice of b. Then we check if this id is in the range $[l_b, r_b]$ of descendants of b.

Since b is known at compile time, testing (3.14) requires exactly the same number of RISC instructions as relative numbering (3.7). Note that the two comparisons in (3.14) are between integers of fixed size, which needs not to be longer than $\log_2 n$ bits. In each of the hierarchies in our data set, 16 bits comparisons are sufficient, and it is extremely unlikely that hierarchies will ever contain more than 2^{32} types. In contrast, subtyping tests in a bit vector encoding scheme may be implemented in only one comparison of bit vectors, but since the length of these vectors is not fixed (e.g., 95 bits for Geode in the NHE scheme), this comparison must be repeated several times.

Also note that the test (3.14) is similar to array bounds checking. Therefore, it may be possible to optimize the implementation on an architecture with dedicated instructions for this kind of check. Such architectures include the Intel 80186+ series (`bound` mnemonic) and the Motorola 680x0 series as well as Motorola 68300 (`chk2` mnemonic).

Palacz and Vitek [110] explain that for reasonably sized hierarchies, including all hierarchies in standard benchmarks, it is possible to implement the check (3.14) using a single jump instruction instead of two. We now give a slightly improved version of the technique they describe. Consider the predicate

$$(x_1 > y_1) \wedge (x_2 > y_2) \tag{3.15}$$

where x_i and y_i, $i = 1, 2$ are 15-bit integers. Then we pack x_1 and x_2 (respectively, y_1 and y_2) together in a single 32-bit integer x (respectively, y). Let $x = 2^{16}x_1 + x_2$, and $y = 2^{16}y_1 + y_2 + M$, where $M = 2^{32} + 2^{16}$. Then, the expression

$$(y - x) \ \textbf{and} \ M \tag{3.16}$$

is zero if and only if (3.15) holds. Checking whether (3.16) is zero requires a subtraction, a bit mask operation and a jump.

3.6.2 Slicing

The essence of slicing is that when the global consecutiveness property cannot be satisfied, we maintain a weaker, local property. More specifically, given a slice $S \subseteq \mathcal{T}$, let $\varphi(S) \subseteq \wp(\mathcal{T})$ be the set of sets of descendants of types in this slice, i.e.,

$$\varphi(S) = \{\text{descendants}(t) \mid t \in S\}.$$

Definition 3.1 *A slice S satisfies the* local consecutiveness property *if there is an ordering of $\mathcal{T}$ in which all members of $\varphi(S)$ are consecutive.*

A partitioning of $\mathcal{T}$ into slices which satisfies the local consecutiveness property always exists, since this property trivially hold for singletons. The local consecutiveness property makes it possible to represent the set of all descendants of any type using merely two integers, and implement every type check as interval inclusion test, as done in (3.14).

Figure 3.7 describes a PQE representation of the running example. The global consecutiveness property holds in this case— only one slice is used—and each type has a single id. To check whether G is a descendant of A, we only need to check whether $\text{id}_{\mathsf{G}} = 4$ falls in the range $[l_{\mathsf{A}}, r_{\mathsf{A}}] = [1, 6]$.

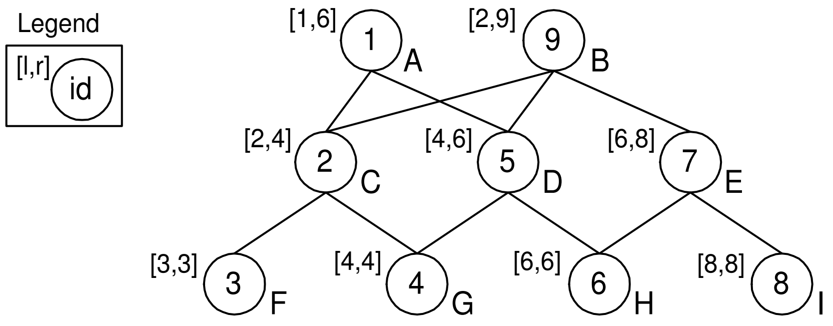


Figure 3.7: PQ-encoding of the hierarchy of Figure 3.1

The numbering of Figure 3.7 was found using a PQ-tree, a data structure that maintains a set of *orderings* (permutations) of some *universe*. Initially, the PQ-tree represents the set of all $9!$ orderings of types A, $\dots$, I. The tree is updated progressively, narrowing down this set, to reflect the constraints that the descendants of all types are consecutive. For each of the types, we try to update the PQ-tree so that it represents only the orderings in which the descendants of this type are consecutive.

In the running example, this update process never fails; we therefore ended in a PQ-tree representation of all orderings which satisfy the global consecutiveness property. The ordering depicted in Figure 3.7 was obtained by picking one of these orderings.

If an ordering which satisfies the global consecutiveness property exists, then our algorithm is guaranteed to find it. In the general case, we use a greedy heuristic for minimizing the total number of slices, and hence the encoding length: "try to make the current slice as large as possible without violating the local consecutiveness property".

Figure 3.8 shows our running example hierarchy augmented with a new type N, added as an additional ancestor of E.

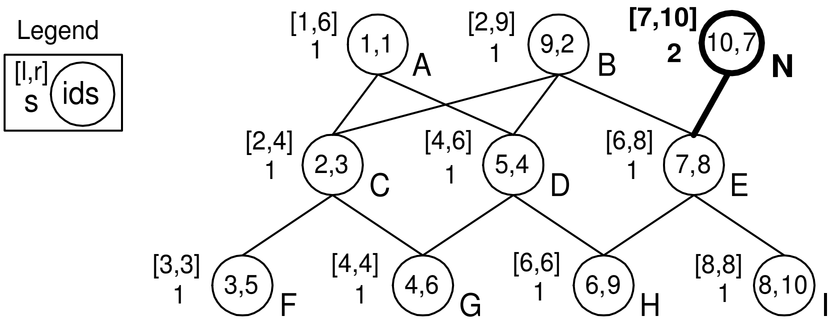


Figure 3.8: A two slices PQ-encoding of the hierarchy of Figure 3.1 augmented with a new type N

There is no ordering of the types in this hierarchy which satisfies the global consecutiveness property. Therefore, PQ-encoding is inclined to use two slices:

$$\begin{aligned} S_1 &= \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}\}, \\ S_2 &= \{\mathsf{N}\}. \end{aligned} \tag{3.17}$$

We see that the greedy heuristic assigns all types but N to the first slice. In Figure 3.8 the slice of each type is written to its left.

Comparing Figure 3.8 with Figure 3.7 we also see that each type has now two id's instead of one. To check whether G is a descendant of N, we first surmise that the slice of N is 2. We therefore use the *second* id of G, $\mathrm{id}_{\mathsf{G}}@\ 2 = 6$ and check whether it falls in the range $[l_{\mathsf{N}}, r_{\mathsf{N}}] = [7, 10]$.

3.6.3 PQ-Trees

PQ-trees were invented by Booth and Leuker [17] [6] who used them to test for the *consecutive 1*'s property in binary matrices of size $r \times s$, in $O(k + r + s)$ time, where k is the number of 1's in the matrix. Booth and Leuker's result gave rise to the first linear-time algorithm for recognizing interval graphs. Later, PQ-trees were used for other graph-theoretical problems, such as on-line planarity testing [12, 13] and maximum planar embeddings [14, 82, 83].

Definition 3.2 *A PQ-tree over a universe $\mathcal{T}$ is either a special $\perp$ symbol, or an ordered tree data-structure with a leaf for every member of $\mathcal{T}$, and such that each internal node is labelled as either a* Q-node *or a* P-node.

A PQ-tree represents a set of orderings of $\mathcal{T}$. The $\perp$ symbol represents an empty set of orderings. Otherwise, each Q-node in the data-structure represents the requirement that all children of the node must occur in the order they occur in the tree or in reverse order. A P-node represents the requirement that these children must occur together, but in no specific order.

The *universal PQ-tree*, denoted $\mathcal{P}^{\top}$ represents the set of all orderings; it has a P-node as a root and a leaf for every member of $\mathcal{T}$. A more interesting example is given by Figure 3.9 which depicts a PQ-tree over the universe $\mathcal{T} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}$. This tree represents the requirement that A, B, and C must occur together, either in this order or in the reverse order $\langle \mathsf{C}, \mathsf{B}, \mathsf{A} \rangle$.

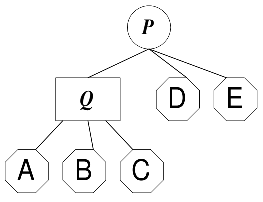


Figure 3.9: A PQ-tree over the universe $\mathcal{T} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}\}$, with a single P-node (depicted as a circle), a Q-node (depicted as a rectangle) and five leaves (depicted as octagons)

Let $\mathrm{consistent}(\mathcal{P})$ denote the subset of orderings of the universe $\mathcal{T}$ which is represented by a PQ-tree $\mathcal{P}$. The specific ordering of $\mathcal{T}$ obtained by a DFS traversal of $\mathcal{P}$, $\mathcal{P} \neq \perp$, is denoted $\mathrm{frontier}(\mathcal{P})$.

[6] In fact, Lempel et al. [92] were the first to coin the term *PQ-expressions*. PQ-trees are nothing more than an efficient representation of PQ-expressions.

In Figure 3.9 we have

$$\mathrm{frontier}(\mathcal{P}) = \langle \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E} \rangle, \tag{3.18}$$

and

$$\begin{aligned}\mathrm{consistent}(\mathcal{P}) = \{ & \langle \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E} \rangle, \langle \mathsf{C}, \mathsf{B}, \mathsf{A}, \mathsf{D}, \mathsf{E} \rangle, \langle \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{E}, \mathsf{D} \rangle, \langle \mathsf{C}, \mathsf{B}, \mathsf{A}, \mathsf{E}, \mathsf{D} \rangle, \\ & \langle \mathsf{E}, \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D} \rangle, \langle \mathsf{E}, \mathsf{C}, \mathsf{B}, \mathsf{A}, \mathsf{D} \rangle, \langle \mathsf{D}, \mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{E} \rangle, \langle \mathsf{D}, \mathsf{C}, \mathsf{B}, \mathsf{A}, \mathsf{E} \rangle, \\ & \langle \mathsf{D}, \mathsf{E}, \mathsf{A}, \mathsf{B}, \mathsf{C} \rangle, \langle \mathsf{D}, \mathsf{E}, \mathsf{C}, \mathsf{B}, \mathsf{A} \rangle, \langle \mathsf{E}, \mathsf{D}, \mathsf{A}, \mathsf{B}, \mathsf{C} \rangle, \langle \mathsf{E}, \mathsf{D}, \mathsf{C}, \mathsf{B}, \mathsf{A} \rangle \}.\end{aligned} \tag{3.19}$$

There are two transformations of a PQ-tree $\mathcal{P}$ which preserve $\mathrm{consistent}(\mathcal{P})$: swapping any two children of a P-node, and reversing the order of the children of a Q-node. PQ-trees $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent ($\mathcal{P}_1 \equiv \mathcal{P}_2$) if $\mathcal{P}_2$ can be reached from $\mathcal{P}_1$ by a series of these transformations. Thus, $\mathrm{consistent}(\mathcal{P})$ can be more formally defined as

$$\mathrm{consistent}(\mathcal{P}) = \{\mathrm{frontier}(\mathcal{P}') \mid \mathcal{P}' \equiv \mathcal{P}\}, \tag{3.20}$$

and $\mathrm{consistent}(\perp) = \emptyset$.

A *constraint* (on orderings) is the requirement that certain elements of the universe occur together. A constraint is represented simply as a subset of the elements of the universe. (In our application, each constraint will be the set of descendants of a given type.) We denote the set of all orderings that satisfy constraint I as $\Pi(I)$.

Consider the special cases $I = \emptyset$, $|I| = 1$, or $I = \mathcal{T}$. Then, it is easy to see that all orderings satisfy I. Thus, in all these cases,

$$\Pi(I) = \mathrm{consistent}(\mathcal{P}^\top).$$

More generally,

Fact 3.3 *For every constraint I there exists a PQ-tree $\mathcal{P}$, $\mathcal{P} \neq \perp$, such that*

$$\Pi(I) = \mathrm{consistent}(\mathcal{P}).$$

PROOF. The root of $\mathcal{P}$ is a P-node whose children are the leaves $\mathcal{T} \setminus I$ and another P-node whose children are the leaves I. $\square$

Let $\mathbf{I} \subseteq \wp(\mathcal{T})$ be a collection of constraints. Then, $\mathbf{\Pi}(\mathbf{I})$ is the set (which may be empty) of orderings that satisfy *all* constraints in $\mathbf{I}$, i.e.,

$$\mathbf{\Pi}(\mathbf{I}) = \bigcap_{I \in \mathbf{I}} \Pi(I).$$

(In our application, each slice S generates a collection of constraints $\varphi(S)$.)

For example, the requirement that types A and B are consecutive, and that types B and C are consecutive, is represented by

$$\mathbf{I} = \{ \{\mathsf{A}, \mathsf{B}\}, \{\mathsf{B}, \mathsf{C}\} \}.$$

It is easy to check that $\mathbf{\Pi}(\mathbf{I})$ is the set (3.19) of orderings consistent with the PQ-tree of Figure 3.9. Another example is the empty set of constraints which is satisfied by all orderings, i.e., $\mathbf{\Pi}(\emptyset) = \mathrm{consistent}(\mathcal{P}^\top)$. More generally,

Theorem 3.4 (BOOTH-LEUKER (1976)) *Suppose that $|\mathcal{T}| > 2$. Then, for every collection of constraints* $\mathbf{I}$ *there exists a PQ-tree* $\mathcal{P}$*, and for every PQ-tree* $\mathcal{P}$ *there exists a collection of constraints* $\mathbf{I}$ *such that* $\mathbf{\Pi}(\mathbf{I}) = \mathrm{consistent}(\mathcal{P})$.

Algorithm 3.1 Compute the PQ-tree of all orderings which satisfy a set of constraints $\mathbf{I}$

Given a universe $\mathcal{T}$, and a set of constraints $\mathbf{I} \subseteq \wp(\mathcal{T})$, return a PQ-tree of all orderings of $\mathcal{T}$ which satisfy $\mathbf{I}$.

```
Procedure genTree(I)
$\mathcal{P} \leftarrow \mathcal{P}^\top$ // $\mathcal{P}^\top$ is the universal PQ-tree.
For all $I \in \mathbf{I}$ do
    $\mathcal{P} \leftarrow$ reduce$(\mathcal{P}, I)$
od
return $\mathcal{P}$
```

Constructively, the tree $\mathcal{P}$ is generated from $\mathbf{I}$ using the iterative process described in Algorithm 3.1. The heart of the algorithm is the procedure reduce which “adds” a constraint to a PQ-tree in a time proportional to the size of the constraint. We here use Booth and Leuker [17] clever implementation of reduce as a black box.[7] Formally,

Theorem 3.5 (BOOTH-LEUKER (1976)) *Given a PQ-tree* $\mathcal{P}$ *and a constraint* I*, the call* reduce$(\mathcal{P}, I)$ *runs in* $O(|I|)$ *time, while the value it returns satisfies*

$$\mathrm{consistent}(\mathsf{reduce}(\mathcal{P}, I)) = \mathrm{consistent}(\mathcal{P}) \bigcap \Pi(I).$$

Note that the set $\mathrm{consistent}(\mathcal{P}) \bigcap \Pi(I)$ may be empty, in which case reduce returns $\perp$.

3.6.4 Finding a PQ-encoding

There are hierarchies for which Algorithm 3.1 can be used to find a PQ-encoding. A case in point is our running example (see Figure 3.7): Each of the nine types in this example imposes a constraint on the permissible orderings. Singleton constraints are not interesting since they are satisfied by any ordering. The remaining constraints are

$$\begin{aligned} I_\mathsf{C} &= \{\mathsf{C}, \mathsf{F}, \mathsf{G}\}, \\ I_\mathsf{D} &= \{\mathsf{G}, \mathsf{D}, \mathsf{H}\}, \\ I_\mathsf{E} &= \{\mathsf{H}, \mathsf{E}, \mathsf{I}\}, \\ I_\mathsf{A} &= \{\mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{A}\}, \\ I_\mathsf{B} &= \{\mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{E}, \mathsf{I}, \mathsf{B}\}. \end{aligned} \tag{3.21}$$

The constraint I_C is that the descendants of type C must occur consecutively, etc. The call genTree$(\{I_\mathsf{C}, I_\mathsf{D}, I_\mathsf{E}, I_\mathsf{A}, I_\mathsf{B}\})$ returns the PQ-tree of Figure 3.10. (Section 3.10 shows and explains the intermediate trees generated in the computation process.)

[7]The curious reader may care to know that reduce conducts a bottom-up traversal of the input tree, applying one of eleven PQ-tree transformations at each step.

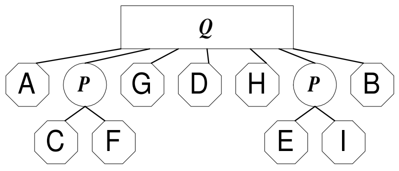


Figure 3.10: The PQ-tree returned from the call genTree on the constraints (3.21)

The PQ-tree of the figure has one Q-node and two P-nodes with two children each. Therefore, this tree represents 8 different orderings, each satisfying the global consecutiveness property. The encoding of Figure 3.7 uses the ordering represented by the tree's frontier $\langle \mathsf{A}, \mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{E}, \mathsf{I}, \mathsf{B}\rangle$.

In the general case, a PQ-encoding may require more than one slice. In such cases, the application of genTree to the set of all constraints $\varphi(\mathcal{T})$ returns $\perp$. (An example can be found in Section 3.10.)

Algorithm 3.2 generates a PQ-encoding for *any* given hierarchy $(\mathcal{T}, \preceq)$.

The main data structure used by the algorithm is the set $\mathbf{S}$, which is an internal representation of the set of slices. Each $\mu \in \mathbf{S}$ is a record $\langle \mathcal{P}, \mathrm{id}\rangle$, where $\mu.\mathcal{P}$ is the PQ-tree of the slice, and $\mu.\,\mathrm{id}$ is the id of the slice. The set $\mathbf{S}$ is discarded after the algorithm computes the encoding.

After a simple initialization (line 1), the algorithm comprises of three stages. The first outer loop (lines 2–15) finds the slices. In this loop we try to find an existing slice for each type, by trying to incorporate (line 4) the constraints that its descendants must lie consecutively, into each of the PQ-trees of the existing slices. If this should fail then we create a new slice (lines 11–14).

The second stage is the loop of lines 16–22, which assigns a unique id to each type with respect to each slice. The last stage (lines 23–26) is to find the interval of the id's of the descendants of each type $a \in \mathcal{T}$, i.e., the id's, with respect to the slice of a, of the right-most and left-most type among the descendants of a.

Lemma 3.6 *Algorithm 3.2 runs in* $O\left(|\mathbf{S}| \cdot |\preceq|\right)$ *time.*

PROOF. The first stage is the slowest. At this stage reduce is invoked at most $|\mathbf{S}|$ times for each of the types in the input. Using Theorem 3.5 the total time of all such invocations is

$$O\Big(|\mathbf{S}| \cdot \sum_{a\in\mathcal{T}} |\mathrm{descendants}(a)|\Big) = O\Big(|\mathbf{S}| \cdot |\preceq|\Big).$$

The second stage runs in time

$$O\Big(|\mathbf{S}| \cdot |\mathcal{T}|\Big) \subseteq O\Big(|\mathbf{S}| \cdot |\preceq|\Big),$$

while the third stage time complexity is

$$O\Big(\sum_{a\in\mathcal{T}} |\mathrm{descendants}(a)|\Big) = O\Big(|\preceq|\Big) \subseteq O\Big(|\mathbf{S}| \cdot |\preceq|\Big). \quad \square$$

Algorithm 3.2 Compute the PQ-encoding of a hierarchy $(\mathcal{T}, \preceq)$

```
1: $\mathbf{S} \leftarrow \emptyset$ // $\mathbf{S}$ *is a set of the slices created so far. Each slice* $\mu \in \mathbf{S}$ *is represented as a*
   // *record* $\langle \mathcal{P}, \mathrm{id} \rangle$*, where* $\mu.\mathcal{P}$ *is the PQ-tree of the slice, and* $\mu.\mathrm{id}$ *is the* id *of the slice.*
2: **For all** $a \in \mathcal{T}$ **do** // *Find a PQ-tree consistent with type* $a$.
3:    **For all** $\mu \in \mathbf{S}$ **do** // *Try to find a slice* $\mu$ *into which* $a$ *could be inserted*
4:       $\mathcal{P}' \leftarrow \mathsf{reduce}(\mu.\mathcal{P}, \mathrm{descendants}(a))$
5:       **If** $\mathcal{P}' \neq \perp$ **then** // *Type* $a$ *can be inserted into slice* $\mu$
6:          $\mu.\mathcal{P} \leftarrow \mathcal{P}'$ // *In the updated PQ-tree* $\mathrm{descendants}(a)$ *are consecutive*
7:          $s_a \leftarrow \mu.\mathrm{id}$ // *Type* $a$ *belongs to slice* $\mu$
8:          **next** $a$ // *Finished handling type* $a$
9:       **fi**
10:   **od**
      // *Type* $a$ *could not be inserted into any of the existing slices*
11:   $\mu \leftarrow$ **new** Slice // *Generate a new slice* $\mu$
12:   $\mu.\mathcal{P} \leftarrow \mathsf{reduce}(\mathcal{P}^\top, \mathrm{descendants}(a))$ // *By Fact 3.3* $\mu.\mathcal{P} \neq \perp$.
13:   $\mu.\mathrm{id} \leftarrow |\mathbf{S}| + 1$ // *Slice* id*'s are allocated in order* $1, 2, \ldots$
14:   $\mathbf{S} \leftarrow \mathbf{S} \cup \{\mu\}$
15: **od**
16: **For all** $\mu \in \mathbf{S}$ **do** // *Assign unique* id*'s to types*
17:    $\mathrm{id} \leftarrow 1$ // *The first unused* id *in the slice* $\mu$.
18:    **For all** $a \in \mathrm{frontier}(\mu.\mathcal{P})$ **do** // *Assign* id*'s to all types with respect to slice* $\mu$
19:       $\mathrm{id}_a@\,(\mu.\mathrm{id}) \leftarrow \mathrm{id}$
20:       $\mathrm{id} \leftarrow \mathrm{id} + 1$
21:    **od**
22: **od**
23: **For all** $a \in \mathcal{T}$ **do** // *Assign an interval to each type* $a$
24:    $\mathbf{D} \leftarrow \{\mathrm{id}_b@\ s_a \mid b \in \mathrm{descendants}(a)\}$;
25:    $[l_a, r_a] \leftarrow [\min(\mathbf{D}), \max(\mathbf{D})]$
26: **od**
```

We do not know of any efficient algorithm for finding the optimal PQ-encoding, i.e., the encoding which achieves the minimal number of slices. This is the reason why Algorithm 3.2 is non-deterministic in the following sense: The order at which types are inserted into PQ-trees (line 2) is unspecified. After having tried several traversal orders, including a random one, we concluded that the differences in the encoding length is small. Our empirical findings indicate that the best results are obtained by a reverse topological-order in which the leaves with the largest number of ancestors are visited first.

Similarly, the order at which we try to find the slices (line 3) is not specified by the algorithm. We found empirically that the best encoding is obtained by trying the slices in the order of decreasing size, i.e., trying the largest slice first, and the smallest one last. A heuristic which gives almost identical results is to try the slices at the order of their creation, with the oldest slice first.

3.7 Optimizations

In this section we describe how Algorithm 3.2 can be further optimized. We have five different, non-language specific, optimization techniques targeted at improving the various complexity measures.

1. *ID Range Compaction* (Section 3.7.1) reduces the space complexity measure, specifically by decreasing the memory footprint of the pseudo-arrays id_a. With this optimization, borrowed from ideas originated by Vitek et al. [133], it is possible to use byte-sized entries for all but the first entry of these arrays.

2. *Pruning Bottom Trees* (Section 3.7.2) targets the encoding creation time measure. We show that the heavy-weight PQ-trees algorithm needs to be run only on the smaller core portion of the input hierarchy.

3. *Reordering Type Records* (Section 3.7.3) is a novel technique which simultaneously improves three complexity measures: space, instruction count and test time. In this optimization, the type records of the runtime environment are pre-sorted in linear time by the first entry of the id pseudo-array. This makes it possible to eliminate this first entry which is (in a sense) encoded by the pointer stored in each object to its type record. A comparison of id's stored in the first entry is replaced by a comparison of these pointers. (The main cost is in the requirement that type records occur in a fixed order, which may be a burden to other parts of the computing environment.)

4. *Heterogeneous Encoding* (Section 3.7.4) also reduces space complexity, by switching to binary matrix encoding in slices which contain no more than 8 types. This optimization which is similar to the one suggested by Vitek et al. [133] may increase the instruction count and the test time complexity measures in subtyping tests involving these slices.

5. *Coalescing ID-Arrays* (Section 3.7.5) is another novel technique which targets the space complexity while increasing the instruction count and the test time. The idea here is that if suffixes of the id pseudo-arrays are identical, they can be shared at the cost of an extra indirection.

3.7.1 ID-Range Compaction

ID-range compaction reduces the encoding length as generated by Algorithm 3.2. Let D be the set of descendants of a slice S:

$$D = \bigcup_{a \in S} \mathrm{descendants}(a).$$

Clearly, $|S| \leq |D|$. However, it is often the case, especially with the smaller slices, that $|S| \ll |D|$, and that $|D|$ is close to n. ID-range compaction relies on the observation that in these cases id's can be reused while numbering the types in D. This reuse makes it possible to use fewer bits for the representation of each id.

The critical point to note is that two types $b_1, b_2 \in D$ need to be assigned distinct identifiers only if there is a type $a \in S$, such that $b_1 \in \mathrm{descendants}(a)$, while $b_2 \notin \mathrm{descendants}(a)$. Phrased differently, S partitions $\mathcal{T}$ into equivalence classes, such that types b_1 and b_2 are in the same equivalence class iff

$$\mathrm{ancestors}(b_1) \cap S = \mathrm{ancestors}(b_2) \cap S. \tag{3.22}$$

These equivalence classes are called the S-*partitioning* of $\mathcal{T}$.

The number of different id's needed to encode a slice S is exactly the number of equivalence classes in the S-partitioning of $\mathcal{T}$. We argue that this number is less than twice the slice size, specifically that there are at most

$$\min(2|S|, |\mathcal{T}|)$$

equivalence classes in the S-partitioning of $\mathcal{T}$. The reason is that the local consecutiveness property ensures that for every $a \in S$ there is an interval I_a which consists the id's of descendants of a. These $|S|$ intervals partition the types in D into at most $2|S| - 1$ segments, such that all types in the same segment can receive the same id. The set $E_0 \equiv \mathcal{T} \setminus D$ defines an additional equivalence class, which is not contained in any interval.

Consider, for example, Figure 3.11, in which the types in D were initially numbered $3, \ldots, 15$.

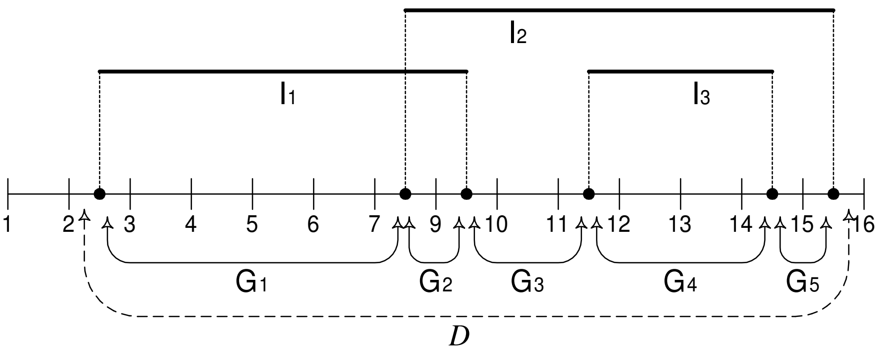


Figure 3.11: Reducing the range needed for PQE

Intervals I_1, I_2 and I_3 drawn in the figure partition D into $5 = 2 \cdot 3 - 1$ segments. This is the maximal possible number of segments, since every type in D must belong to at least one interval. The equivalence classes in this example are $E_0 = \{1, 2, 16\}$, $E_1 = \mathsf{G}_1$, $E_2 = \mathsf{G}_2$, $E_3 = \mathsf{G}_3 \cup \mathsf{G}_5$, and $E_4 = \mathsf{G}_4$.

In all hierarchies in the data set, we found that all slices, except the first, were of size 128 or less. Thus the integral range required for numbering is at most 256 and id_a can be represented as a byte array, with each slice adding a single byte to the encoding length. The first slice receives some special handling as will be described below in Section 3.7.3.

It is possible to modify Algorithm 3.2 to ensure that all but one (the first) slice has their range bounded by 256. Specifically, line 5, must not only check $\mathcal{P}'$, the PQ-tree returned by the reduce routine, but also make sure that the range required for numbering does not exceed 256.[8] Storing the current required numbering range of a PQ-tree, and updating it with each reduce is straightforward. One can also manage the equivalence classes of all slices incrementally in $O(|\preceq|)$ total time.

3.7.2 Pruning Bottom Trees

Recall that in Section 3.4 we defined the core of a multiple inheritance hierarchy $\mathcal{T}' \subseteq \mathcal{T}$, such that $t \in \mathcal{T}'$ if t has a descendant with more than one parent. The set $\mathcal{T} \setminus \mathcal{T}'$ is a collection of bottom-trees discovered in a bottom-up traversal of the hierarchy. Intuitively, the core is where the intricacies of multiple inheritance occur. The bottom-trees are a forest of single inheritance hierarchies, hanging at the bottom of the core.

By pruning in a preprocessing stage all bottom-trees, we reduce the run time of Algorithm 3.2. A lighter machinery is then used to produce the encoding of the bottom-trees. Let $S_1, \ldots, S_k$ be the slices of $\mathcal{T}'$ found in the PQ-encoding of the pruned hierarchy, and $\pi'_1, \ldots, \pi'_k$ be the orderings of $\mathcal{T}'$ with respect to each slice. Thus, π'_i, $i = 1, \ldots, k$ is the ordering defined by the id's of all types with respect to slice S_i. Formally, π'_i satisfies the constraints $\varphi(S_i)$.

Next we describe how to extend π'_i of $\mathcal{T}'$ into an ordering π_i of $\mathcal{T}$ in such a way that it will satisfy the constraints $\varphi(S_i \cup (\mathcal{T} \setminus \mathcal{T}'))$. Consider an arbitrary bottom-tree whose root is t. Since t is not in the core, it has a single parent t', i.e., $\mathrm{parents}(t) = \{t'\}$. Type t' must be in the core, otherwise t would not be the root of the bottom tree. (Note that t' might have several other children which are roots of other bottom trees.) When extending the ordering π'_i of $\mathcal{T}'$, we insert the relative numbering ordering of this bottom-tree immediately after (or before) $\pi'_i(t')$.

Figure 3.12 gives an example of the insertion of relative-numbering orderings into the ordering of the core. Figure 3.12a shows the core of the hierarchy of the running example, whereas the bottom-trees are highlighted in bold in Figure 3.12b.

Figure 3.12a shows an ordering π' of the core $\mathcal{T}'$ which satisfies the constraints $\varphi(\mathcal{T}')$,

$$\pi' = \langle \mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{E}, \mathsf{B} \rangle.$$

Figure 3.12b shows the extended ordering π of $\mathcal{T}$ which satisfy the constraints $\varphi(\mathcal{T})$:

$$\pi = \langle \mathsf{A}, \mathsf{N1}, \mathsf{C}, \mathsf{N2}, \mathsf{N5}, \mathsf{N6}, \mathsf{N3}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{N7}, \mathsf{E}, \mathsf{N4}, \mathsf{B} \rangle.$$

Note that the resulting ordering π_i of $\mathcal{T}$ satisfies the old constraints in $\varphi(S_i)$ (since descendants in a bottom-tree are adjacent to their parent in the core) and the new constraint in $\varphi(\mathcal{T} \setminus \mathcal{T}')$ (since relative-numbering ordering satisfies these constraints).

In order to complete the process of incorporating the bottom-trees into a PQ-encoding of the core, we must also assign each of the types in the bottom-trees into a slice. The fact that we inserted the relative numbering ordering of each bottom-tree makes it possible to chose any slice we want for each type in a bottom-tree. We chose to use the first slice for all these types since ID-range compaction works best when the first slice is much larger

[8]Note that this does not necessarily happen when the slice size hits 128.

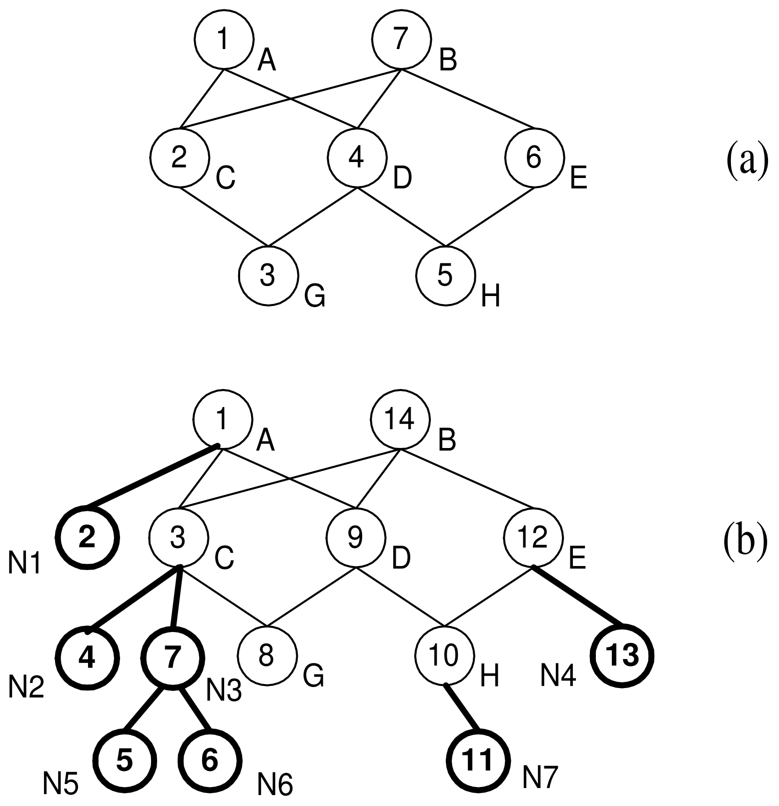


Figure 3.12: PQE of the core of the running example (a) and PQE after inserting some bottom-trees (b)

than the others. Another motivation for this choice is the "reordering of type records" optimization which, as we shall see below, makes it possible to eliminate all bits used for the first slice.

3.7.3 Reordering Type Records

Consider again the subtyping test $a \preceq b$. So far it was assumed that the type a is *given* at run time. In reality, however, an object o is given and the runtime system must first infer its type a. Typically o stores a pointer p_a to its *type record*, a memory block with run time representation of a. The various encoding schemes store their auxiliary information in this area. Many object-oriented language implementations mandate other uses to the type records, including dispatching, downcasting, serialization, and garbage collection.

The *reordering type records* optimization technique makes use of the degree of freedom the compiler has in placing type records in memory.[9] The simplest application of this technique is to relative numbering (Section 3.5): Type records are placed in memory in the same order as postorder traversal of the type hierarchy. In doing so, the pointer p_a plays the role as the ordinal in the postorder r_a. As a result, the encoding length is reduced

[9]We make the natural assumption that the location of the encoding tables is in a protected location of memory which is not subject to garbage collection. The reason is that these tables are generated as part of the compilation process and are not changed at runtime.

to zero and one load instruction in the subtyping test is saved.

Similarly, in range-compression (3.13), p_a replaces the global id_a. If specialization is used then we obtain an encoding scheme with zero encoding length, but non-constant test time and instruction count.

We do not know whether the technique is applicable to either bit-vector encodings or to Cohen's algorithm and its generalizations, PE and BPE. However, in PQE, the ability to reorder type records makes it possible to eliminate entirely the id's of types with respect to the first slice. Specifically, $\mathrm{id}_a @ 1$ of a type a is encoded in the pointer p_a. The saving is significant since the first slice occupies the largest number of bits. This technique also saves one load instruction when type b belongs in the first slice. Since the first slice constitutes around 90% of the types, we expect this saving to lead to a noticeable saving in the average test time.

We finally note that this technique is applicable even with the unique C++ object layout. In this layout [68] an object may contain several pointers to several *distinct* type records (VTBLs in the C++ jargon).

The reason that we can encode integers in pointers even though there is no unique value p_a for a type a is that the subtype tests of relative numbering (3.7), range compression (3.13), and PQE (3.14), all check for *inequality* rather than equality. We simply allocate a range of memory addresses to all type records of a given type, rather than a single address, as the value r_a (as in (3.7)) or the id (as in (3.13) and (3.14)).

3.7.4 Heterogeneous Encoding

Heterogeneous encoding is yet another optimization targeted at reducing the encoding length. Recall that in the binary matrix each type adds exactly one bit to the encoding of all other types. The PQ-encoding of a small slice with $k < 8$ types adds a byte to the array id_a of each other type a, which is less efficient than using the binary matrix for types in this slice. In heterogeneous encoding, subtyping tests $a \preceq b$, where b belongs in such a small slice, are implemented using the binary matrix. Since b is known at compile time, the compiler can choose the appropriate code to plant at the subtyping test. We found that heterogeneous encoding may give rise to significant improvement to the encoding length. On the other hand, the total number of types in small slices is negligible, and therefore we do not expect a noticeable impact on the instruction count and test time.

3.7.5 Coalescing ID-Arrays

We now turn to describing *Coalesced PQ-Encoding* (CPQE). This memory optimization is based on the observation that the contents of the pseudo-arrays id_a tend to be similar. We rely on the fact that the first entry of these arrays is represented implicitly. Let id'_a denote the array obtained from id_a by truncating its first entry. Then, many of the arrays id'_a are identical, and need to be stored only once.

More specifically, we claim that the number of distinct arrays id' is exactly the number of equivalence classes in G-partitioning of $\mathcal{T}$, where $G = \mathcal{T} \setminus S_1$. In other words, two types a, b are in the same equivalence class iff $\mathrm{id}'_a = \mathrm{id}'_b$. Formally,

Lemma 3.7 *Let a, b be two types, and $G = \mathcal{T} \setminus S_1$. Then*

$$\mathrm{ancestors}(a) \cap G = \mathrm{ancestors}(b) \cap G \Leftrightarrow \mathrm{id}_a@\ i = \mathrm{id}_b@\ i \ \text{ for } 2 \le i \le k.$$

PROOF. We previously showed (3.22) that two types can have the same identifiers if and only if they are in the same equivalence class, i.e.,

$$\mathrm{id}_a@\ i = \mathrm{id}_b@\ i \Leftrightarrow \mathrm{ancestors}(a) \cap S_i = \mathrm{ancestors}(b) \cap S_i.$$

Since $S_2 \ldots S_k$ partition G we have that

$$\begin{aligned} &\mathrm{ancestors}(a) \cap G = \mathrm{ancestors}(b) \cap G \Leftrightarrow \\ &\mathrm{ancestors}(a) \cap S_i = \mathrm{ancestors}(b) \cap S_i \ \text{ for } 2 \le i \le k \Leftrightarrow \\ &\qquad \mathrm{id}_a@\ i = \mathrm{id}_b@\ i \ \text{ for } 2 \le i \le k. \quad \square \end{aligned} \tag{3.23}$$

Furthermore, the number of distinct arrays id' is always smaller or equal to the size of the core. (The core is the set of types not belonging to a bottom tree; See Section 3.4.) Recall that the bottom trees were added to the first slice after they were pruned (see Section 3.7.2). Since each type in a bottom tree has the same ancestors set as the root of that tree, they are in the same equivalence class, and therefore can be coalesced together.

CPQE uses a bucket sort to find the distinct values of arrays id' in linear time, and then represents each type a as a pointer p'_a to one of these distinct values. The cost of the coalesced representation is in another level of indirection for subtyping tests involving the second or higher slice.

The pointer p'_a is not stored as an absolute memory address but rather as an index of an array Z, whose entries are the distinct id' arrays. Also the degree of freedom in placing entries in Z, is employed to encode $\mathrm{id}_a@\ 2$ (id's of the second slice) in p'_a in the same fashion that $\mathrm{id}_a@\ 1$ was encoded as p_a.

In the test $a \preceq b$, if it is found that b belongs in slice S_2, then instead of using $\mathrm{id}_a@\ 2$ in the test (3.14), the compiler emits code for comparing p'_a with the values l_b and r_b, which are, as usual, specialized into the test code. The entries in array Z are then the arrays id'' produced by truncating the first two entries of the arrays id.

A strong incentive to use CPQE is raised by languages such as C++, in which objects may contain *multiple* pointers to several *distinct* type records [68]. Since these type records are similar, but not identical, the implementer must choose between (i) *replicating* the subtyping encoding data in each such record, or (ii) *sharing* at the cost of another level of indirection during subtyping tests. Coalescing optimization may tip the scale towards the sharing alternative.

3.8 Results

Having described different optimization techniques we would like to appreciate the trade-offs offered by these. To do so, we define (Section 3.8.1) variants of the main encoding scheme. We then show (Section 3.8.2) how the encoding length of these variants depends on the output of our main algorithm (Algorithm 3.2), and in particular the number of slices and the distribution of their size. Section 3.8.3 compares the encoding length achieved by the different variants with the achievements of previous work. Section 3.8.4 gives the results of our timing of the algorithm for computing the encoding length.

3.8.1 Variants of the PQ-Encoding Scheme

There are many variants of PQ-encoding, depending on which of the optimizations described in the previous section are applied. The first two optimizations: ID range compaction and bottom tree elimination, which do not add to the main complexity measures are in fact incorporated to the main algorithm. We next define three encoding variants which successively apply the three other optimizations:

1. Regular PQ-encoding, or RPQE for short is the variant in which reordering the type records is used to eliminate the representation of the first slice from the id arrays.

2. The principal acronym PQE is reserved to the variant which also applies the heterogenous encoding optimization. As explained above, the cost is in longer subtyping tests in the rare cases involving the smaller slices.

   Thus, in PQE, there are three kinds of slices: The first slice, whose representation is eliminated thanks to reordering of type-records. Heterogeneous encoding based on the binary matrix representation is used for slices whose size is smaller than 8. Each of the remaining slices occupies a single byte in the array id, which is used in the basic subtyping tests of PQE (3.14).

3. CPQE is the encoding variant obtained from PQE by applying in addition the remaining fifth optimization: coalescing of ID-arrays, which adds to the cost of subtyping tests involving the third or higher slice.

3.8.2 Output of the PQ-Algorithm

Algorithm 3.2, the main algorithm behind the PQ-encoding, returns a partitioning of the hierarchy into slices. It was mentioned before that the size of slices vary widely. Using the hierarchies in our data set we now turn to studying this variety in detail.

Table 3.2 displays some of the essential parameters of the slice size distribution. These parameters will become useful in appreciating the algorithm performance and the trade-offs offered by the different optimizations. We can also use these to calculate the encoding length of the three encoding schemes described above.

Even though we do not have a non-trivial upper bound on the number of slices, the second column of the table shows that in actual hierarchies, k, the number of slices, is often small, and it does not increase as quickly as n. Thus, we have reasons to believe that $O(kn)$, the asymptotic space complexity of algorithm Algorithm 3.2, is closer to linear than quadratic. Similar conclusions can be drawn on $O(k|\preceq|)$, the time complexity of the algorithm.

Hierarchy	k[a]	n_1/n[b]	k_2[c]	n_2[d]	n_2/n	m[e]
IDL	1	100.0%	0	0	0.0%	0
JDK 1.1	2	99.6%	1	1	0.4%	1
Laure	2	98.0%	1	6	2.0%	7
Ed	10	87.8%	7	20	4.6%	145
LOV	12	86.2%	9	26	6.0%	164
Unidraw	2	99.7%	1	2	0.3%	2
Cecil	5	94.1%	2	6	0.6%	101
Geode	16	86.0%	8	24	1.8%	419
JDK 1.18	6	97.5%	3	9	0.5%	74
Self	13	97.2%	11	31	1.7%	63
Eiffel4	11	89.1%	3	9	0.5%	376
JDK 1.22	8	97.6%	4	12	0.3%	235
JDK 1.30	8	97.7%	4	17	0.3%	286

[a]number of slices
[b]fraction of types in the first slice
[c]number of small slices
[d]total number of types in small slices
[e]number of distinct id' arrays

Table 3.2: Some characteristics of the slice partitioning of the PQ algorithm

Integer k is also useful in computing the encoding length of RPQE. Recall that with the exception of the first slice, the id's with respect to each slice can be represented in a single byte. Therefore, the encoding length of RPQE is $8(k-1)$. (Also, consider a variant of RPQE in which type records are not reordered. Then, the encoding length in this variant is $16 + 8(k-1) = 8(k+1)$.)

The next column in the table gives the ratio n_1/n, where n_1 is the number of types in the first slice (which is also the largest slice). We see that in all hierarchies over 85% of the types fall in this slice. In fact, in more than half the hierarchies, this slice occupies at least 97.5% of all types. Thus, we expect that an overwhelming portion of the actual subtyping tests will use this slice. The test time of these will greatly benefit from reordering of type records.

Small slices, i.e., slices with no more than 8 types, receive special handling by PQE. The heterogeneous encoding optimization specifies that types in these slices use a binary matrix representation. The subtyping test then involves bit operations, and is not as simple as the range testing used for the other slices.

The fourth column of Table 3.2 shows k_2, the number of small slices. We see that most of the slices generated by the PQ-algorithm are small. However, examining the next column (the total number of types in the small slices n_2, $k_2 \leq n_2 \leq 8k_2$), we see that n_2 is small. The penultimate column of the table shows that the fraction of types in small slices is tiny, typically less than 1%. We are lead to hope that the frequency of the more complex tests will be equally negligible.

Interestingly, the values shown in Table 3.2 can be used to compute the encoding length of PQE. Since all slices except the first and small slices occupy a single byte in id-

array, we have that this length is

$$8(k - k_2 - 1) + n_2.$$

To compute the encoding length of CPQE we need the final column of the table which shows m, the number of distinct id' arrays. We see that this number is much smaller than the number of types. In fact, $m \leq 256$ in all hierarchies except for Eiffel4, Geode, and JDK 1.30. The pointer p'_a can thus often be represented as a single byte. More generally, the precise encoding length of CPQE is

$$8\left\lceil \frac{\log m}{8} \right\rceil + \frac{(8(k - k_2 - 2) + n_2) \times m}{n}.$$

3.8.3 Encoding Length in the Data Set

Table 3.3 compares the encoding length in bits of the three encoding variants with that of other encoding schemes.

Hierarchy	**CPQE**	**PQE**	**RPQE**	NHE	BPE	PE	DAG[a]	Closure [b]	Binary matrix
IDL	8	0	0	17	32	96	7	27	66
JDK 1.1	8	1	8	19	32	64	9	26	225
Laure	8	6	8	23	63	128	10	74	295
Ed	17	36	72	54	94	216	15	72	434
LOV	21	42	88	57	94	216	16	77	436
Unidraw	8	2	8	30	63	96	8	31	613
Cecil	10	22	32	58	94	192	13	65	932
Geode	39	80	120	95	157	408	21	154	1,318
JDK 1.18	9	25	40	39	94	128	13	48	1,704
Self	9	39	96	53	126	344	12	329	1,801
Eiffel4	27	65	80	72	157	312	15	97	1,999
JDK 1.22	10	36	56	62	157	184	16	57	4,339
JDK 1.30	18	41	56	65	188	216	16	57	5,438

[a]Computed idealistically as $(|\prec_{\mathrm{d}}| \cdot \lceil \log n \rceil)/n$.
[b]Computed idealistically as $(|\preceq| \cdot \lceil \log n \rceil)/n$

Table 3.3: The encoding length of different algorithms

The most important conclusion to draw from the table is that in all hierarchies in the data set, the encoding length achieved by PQE is better than that of all other encoding schemes. The only exception to these is an idealistic DAG representation, in which, as mentioned above, test time can be $O(n)$.

We stress again that the memory requirements of PQE is zero for all single inheritance hierarchies. As can be seen in the table, zero memory footprint occurs even in IDL, which is multiple inheritance. The median improvement over the next best algorithm, NHE, is by 37%, while the average improvement is 50%.

PQE remains the shortest encoding even if it is not optimized by reordering type records (in which case the encoding length increases by 16): Without this optimization, PQE is better than NHE in 9 out of the 13 hierarchies. In one hierarchy (LOV), the encoding length of NHE is 1 bit shorter than PQE, in two hierarchy (Self and JDK 1.18) it is 2 bits shorter, and in one hierarchy (Eiffel4) it is 9 bits shorter.

In comparing PQE with NHE we must also recall that the test time in the bit vector based NHE is non-constant. Thus, even if the two schemes use the same number of bits, subtyping tests in PQE are likely to be more efficient since they do not need to access all bits in the representation of the compared types.

The space reduction of PQE over BPE, the best previous *constant time encoding*, is even more impressive: In the Eiffel4 hierarchy BPE total space requirement is 39KB, compared with 16KB in PQE. These differences are significant since subtyping tests are very frequent. Vitek [110] benchmarks give 320,000 tests in a second. Smaller encoding makes it possible to fit the entire representation in the cache.

Examining the second and third columns of Table 3.3 we see that coalescing of id records, employed by CPQE, shortens the encoding length of PQE, by factors ranging between 2 and 4.3. In fact, CPQE competes favorably even with the idealized DAG encoding!

Hierarchies IDL, Laure, Unidraw and JDK 1.1 are anomalous in the sense CPQE gives a longer encoding than PQE. This phenomenon is explained by the fact that the two-level structure employed by CPQE requires at least 8 bits for p'_a.

We finally note that even RPQE competes favorably with NHE, winning in 7 out of the 13 hierarchies in the data set.

3.8.4 Encoding Creation Time

Table 3.4 compares the encoding creation time of PQE with that of NHE and PE. The creation time of RQPE and CPQE is the same as PQE, and the creation time of BPE is the same as PE.

The comparison is not easy, since the algorithms were run on different machines. Algorithm 3.2 was written in C++ based on the PQ-tree implementation of Leipert [91]. More experimentation is required before a faithful and fair comparison is possible. It appears as if PQE, which is based on a linear algorithm, outperforms the quadratic NHE algorithm. PE, which use a fast implementation of set unions and intersections using bit-vector operations, seems to be the fastest. The Geode hierarchy is toughest for PQE and NHE. In this hierarchy, the average time for processing a type is less than one millisecond in PQE. In all benchmarks the time for computing PQE is less than a second.

3.9 Conclusions and Future Research

The PQE algorithm improves the encoding length, creation time, test time and instruction count of NHE, the most space-efficient previously published encoding algorithm. The CPQE variant reduces the encoding length even further at the cost of an extra indirection in some, typically infrequent, subtyping tests.

Hierarchy	(R | C)PQE [a]	NHE [b]	(B)PE [c]
IDL	1	-	5
JDK 1.1	1	19	10
Laure	4	21	9
Ed	77	136	12
LOV	95	168	10
Unidraw	1	93	10
Cecil	50	-	13
Geode	668	1,902	28
JDK 1.18	29	-	26
Self	122	1,367	22
Eiffel4	299	-	29
JDK 1.22	140	-	77
JDK 1.30	187	-	90

[a] 266 Mhz Pentium II
[b] 500 Mhz 21164 Alpha
[c] 750 Mhz Pentium III, user time in Linux

Table 3.4: Encoding creation time in milliseconds of different algorithms

The main problem which this chapter leaves open is an incremental algorithm for the subtyping problem, as required by languages such as JAVA, in which types may be added as leaves at run time. (Section 4.8 in the next chapter presents such an incremental algorithm.) It turns out that the PQ-data structure is not susceptible to efficient updates of this sort.

On the theoretical side, it would be very interesting to see any non-trivial lower bound for the encoding length.

An interesting instance of the subtyping problem occurs when the ordinary type hierarchy is compounded by an interplay with *genericity*, as in EIFFEL and in the proposed addition of generics to JAVA. In EIFFEL, a double ended queue of rectangles is a subtype of a queue of polygons ($\mathsf{DQueue}[\mathsf{Rectangle}] \preceq \mathsf{Queue}[\mathsf{Polygon}]$) since (i) $\mathsf{Rectangle} \preceq \mathsf{Polygon}$, and (ii) the generic class $\mathsf{DQueue}[T]$ inherited from $\mathsf{Queue}[T]$. EIFFEL has a default subtyping rule which can be written as

$$\forall a, b, A \bullet a \preceq b \Rightarrow A[a] \preceq A[b],$$

and the definition of generic classes which inherit from others adds other rules such as

$$\begin{aligned} \forall a \bullet A[a] &\preceq B[a], \\ \forall a, b \bullet C[a, b] &\preceq D[a[b]]. \end{aligned}$$

The research question is whether pre-processing of such rules can make it possible to decide subtyping more efficiently.

3.10 A Detailed PQ-Tree Example

The example below will shed some light on the “magic” behind Theorem 3.5 and the implementation of reduce due to Booth and Leuker [17].

We first trace the execution of genTree (Algorithm 3.1) where the input is the constraints (3.21) of the running example. The algorithm starts with a universal PQ-tree $\mathcal{P}^\top$ over the universe

$$\mathcal{T} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{H}, \mathsf{I}\},$$

and iteratively calls reduce for each of the input constraints in the order they appear in (3.21). The output of the algorithm is then the PQ-tree depicted in Figure 3.10, which satisfies these constraints. (Using any other order would have resulted in an equivalent PQ-tree.)

Figure 3.13 shows the PQ-tree at each of the intermediate steps in this iterative process. Each subfigure shows the next input constraint (variable I in Algorithm 3.1), and the current PQ-tree (variable $\mathcal{P}$ in the algorithm), where the leaves corresponding to types constrained to appear together in the next iteration are highlighted. Thus, Figure 3.13b is the PQ-tree obtained by performing reduce$(\mathcal{P}^\top, I_{\mathsf{C}})$, while figures 3.13c, d, e, f show the PQ-tree after reducing it with constraints I_{D}, I_{E}, I_{A}, and I_{B}, respectively.

Imposing the constraint $I_{\mathsf{C}} = \{\mathsf{C}, \mathsf{F}, \mathsf{G}\}$ on the initial universal tree (Figure 3.13a) yields the tree of Figure 3.13b, which uses an extra P-node to ensure that these three types occur together. The next constraint to add is $I_{\mathsf{D}} = \{\mathsf{G}, \mathsf{D}, \mathsf{H}\}$. Since type G is common to both I_{C} and I_{D} we have that the permissible orderings must have a subsequence which matches one of the following two patterns:

1. Types C and F occur together, in any order, then type G, and then types D and H together, but in any order.

2. Types D and H occur together, in any order, then type G, and then types C and F together, but in any order.

These two patterns are captured by the PQ-tree of Figure 3.13c, in which one P-node forces C and F to occur together, while another P-node forces D and H to occur together. The Q-node of this tree makes sure that G falls between the pairs $\{\mathsf{C}, \mathsf{F}\}$ and $\{\mathsf{D}, \mathsf{H}\}$.

The transition between Figure 3.13c and Figure 3.13d is even more interesting. Let α_c be the subtree rooted at the Q-node of Figure 3.13c. Then, subtree α_c ensures that the five types C, F, G, D and H occur together. To this requirement we now must add the constraint $I_{\mathsf{E}} = \{\mathsf{H}, \mathsf{E}, \mathsf{I}\}$, which means that H must be adjacent to either E or I. Therefore, H must occur in a boundary position (either first or last) in the placement of the five types in α_c. The problem is that α_c allows D to take the place of H in this boundary position. The remedy is in “lifting” both H and D to the containing Q-node, making sure that if H is first, then D is second, while if H is last then D is in the penultimate position. After having guaranteed that H is in a boundary position, procedure reduce incorporates a P-node of types E and I into the boundary of α. The result is shown in Figure 3.13d.

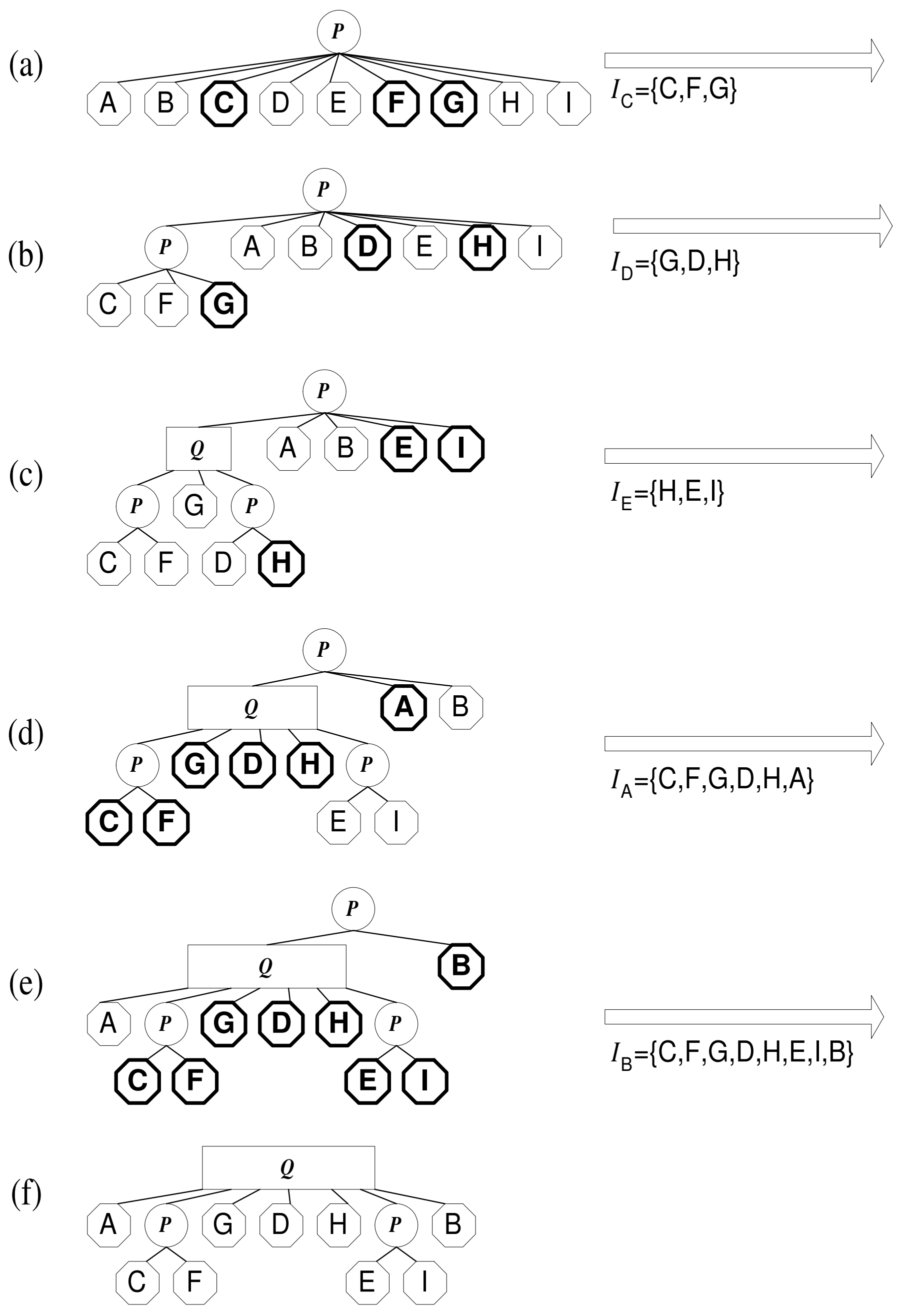


Figure 3.13: Intermediate PQ-trees in the invocation of genTree on the constraints of the hierarchy Figure 3.1

The transition from Figure 3.13d to Figure 3.13e is rather simple. Let α_d be the subtree rooted at the Q-node of Figure 3.13d. Then, the constraint $I_{\mathsf{A}} = \{\mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{A}\}$ is almost satisfied by α_d; the only missing requirement is that α_d does not guarantee that A is adjacent to the others in the requirement. Procedure reduce then makes the leaf A a child of this Q-node. It is possible to do so, since the set $\{\mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}\}$ has a “free” boundary (the other boundary is constrained to be either E or I.

The transition from Figure 3.13e to Figure 3.13f follows the same lines as the previous transition. Again, the set $\{\mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{E}, \mathsf{I}\}$ has only one “free boundary” in the Q-node of Figure 3.13e. The constraint $\{\mathsf{C}, \mathsf{F}, \mathsf{G}, \mathsf{D}, \mathsf{H}, \mathsf{E}, \mathsf{I}\}$ is realized by adding B in the Q-node at this free boundary. Figure 3.13f (which is the same as Figure 3.10) is the final PQ-tree, representing the eight different orderings which satisfy the constraints in (3.21).

To see a situation in which Algorithm 3.1 returns $\perp$, which will make it necessary to use more than one slice, consider the hierarchy depicted before in Figure 3.8. This hierarchy is identical to the running example except that a new type N was added as a parent of type E. This new node adds the constraint that all of its descendants must lie together, i.e., the constraint

$$I_{\mathsf{N}} = \{\mathsf{N}, \mathsf{E}, \mathsf{H}, \mathsf{I}\}, \tag{3.24}$$

is added to **I**.

Figure 3.14 shows the PQ-tree of the augmented hierarchy after all the *other* constraints in (3.21) were incorporated. (This tree is easily obtained by adding type N to the PQ-tree of Figure 3.13f.)

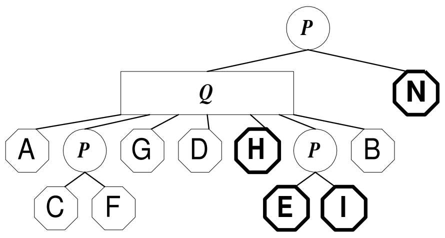


Figure 3.14: PQ-tree with a new configuration in which reduce will return $\perp$

Consider now the constraint (3.24), depicted by highlighting types N, E, H, I in Figure 3.14. By examining the figure, we see that N cannot be made adjacent to any of the types E, H, I. For example, N cannot be adjacent to H, because H lies between D, and one of E and I. In other words, the set $\{\mathsf{H}, \mathsf{E}, \mathsf{I}\}$ has no "free" boundaries. Therefore, calling reduce with the PQ-tree of Figure 3.14 and the constraint (3.24) returns $\perp$.

Chapter 4

Fast Algorithm for Creating Space Efficient Dispatching Tables with Application to Multi-Dispatching

Chapter Summary
The dispatching problem can be solved very efficiently in the single inheritance setting. In this chapter we show how to extend one such solution to the multiple inheritance setting. This generalization comes with an increase to the space requirement by a small factor of κ. This factor can be thought of as a metric of the complexity of the topology of the inheritance hierarchy.

On a data set of 35 hierarchies totaling some 64 thousand types, our dispatching data structure, based on a novel type slicing *technique, exhibits very significant improvements over previous dispatching techniques, not only in terms of the time for creating the underlying data structure, but also in terms of total space used.*

The cost is in the dispatching time, which is no longer constant, but doubly logarithmic in the number of types. Conversely, by using a simple binary search, dispatching time is logarithmic in the number of different implementations. In practice dispatching uses one indirect branch and, on average, only 2.5 binary branches.

Our results also have applications to the space-efficient implementation of the more general problem of dispatching multi-methods.

A by-product of our type slicing technique is an incremental *algorithm for constant-time* subtyping tests *with favorable memory requirements. (The incremental version of the subtyping problem is to maintain the subtyping data structure in presence of additions of types to the inheritance hierarchy.)*

Message dispatching stands at the heart of object-oriented (OO) programs, being the only way objects communicate with each other. Indeed, it was demonstrated [48] that OO programs spend a considerable amount of time in implementing dynamic dispatching. There is a large body of research dedicated to the problem of "efficient" implementation of message dispatching [31, 37, 44–50, 60, 78, 86, 100, 102, 118, 130–132, 135]. The principal optimization objective adopted by most of this prior research was a compact representation of the dispatching data structure, while maintaining a small, preferably constant, dispatch time. A heavy toll incurred in many cases was the time required for creating the dispatching data structure.

This research revisits the problem, trying to optimize another important complexity metric, *creation time*, i.e., the time required for generating the dispatching data structure. Our motivation is the staggering importance of dynamic compilation and re-compilation systems, as found in JAVA. Previous work tended to be conceptually locked in the static compilation model, with few reports on creation time values; when reported, these times were measured in seconds for modest size hierarchies.

Our novel *type slicing* technique gives rise to a very fast algorithm for creating space efficient dispatching data structure. The creation time is improved by one, two and sometimes three orders of magnitude compared to the famous *row displacement* (RD) algorithm [47]. In a collection of 35 hierarchies, totaling over 60,000 types, the slowest runtime of our algorithm was less than a third of a second on a modern processor; this time was on a hierarchy of circa nine thousand types and fourteen thousand messages. In the vast majority of the hierarchies, the creation time was less than a hundredth of a second. Its *space requirement* improves those of RD (arguably the best previously published algorithm in this category), in 32 out of the 35 hierarchies of our data set; the median reduction in space is by a factor of 2.6.

The improvement of creation time and of space requirement comes with a penalty of a small increase to dispatching time. Specifically, dispatching requires a binary search in which the number of branches is logarithmic in the number of implementations of the dispatched message, or alternatively, doubly-logarithmic in the number of types. Each dispatch requires about 2.5 branches on average as well as one dereferencing operation. These numbers may be compared with the two dereferencing steps required by the *Virtual Function Tables* (VFT) [57] standard implementation strategy of C++ [124] in the single inheritance setting. Note that in contrast with our results and most other dispatching algorithms, the VFT technique is valid only in *statically typed* languages [132]. Some dispatching schemes, such as RD and selector coloring (SC), require additional space and one more comparison at runtime in order to work in *dynamically typed* languages.

Interestingly, there is a strong practical evidence that binary searches, which are used in our implementation, may be faster than the simple VFT implementation. The trick is to inline the binary search by generating what was called "static branch code" by the implementors of the SmallEiffel compiler [135], instead of the more general binary search routine. It was shown that with this optimization a binary search between fewer than 50 results was faster than the VFT implementation in most architectures.

One of the explanations of this phenomenon is that indirect branches do not schedule well on modern processors [46, 48–50]. Other, less direct, advantages of inlined binary search is that it can take better advantage of type inference and that it is more susceptible to inlining of method code and any ensuing optimization. The cost of inlining is (of course) in an increase to the code size. Note that several other previous publications suggested using a combination of binary searches, array look-ups, and even linear searches [3, 27, 78, 102] for dispatching.

Informally, we can say that our algorithm generalizes the linear space *interval containment* algorithm [60,100] which is restricted to the single inheritance setting. Our main theoretical result is that the generalization to the multiple inheritance case comes with a κ (the number of slices) factor increase of space. This factor depends only on the *topology of the multiple inheritance hierarchy*, and can be thought of as a metric of its complexity. In practice this factor is small, but in arbitrary hierarchies it might be in the order of the number of types.

In all single inheritance hierarchies, $\kappa \equiv 1$. We provide a heuristic for finding an upper bound of κ, and an actual implementation of the generalization. In our data set of 19 multiple inheritance hierarchies the median value of κ is 6.5, the average is 7.3, and the maximum is 19. We stress that the space increase is by a factor of at most κ; in practice, we find much better results.

Our dispatching technique has also applications to space-efficient implementation of *multi-dispatching*.

Finally, our type slicing technique also provides an *incremental* algorithm for constant-time *subtyping tests* with favorable memory requirements. We provide theoretical analysis of our algorithm, as well as practical evidence that our algorithm is fast even when compared to previous non-incremental algorithms.

Outline The remainder of this chapter is organized as follows. The dispatching problem is defined in Section 4.1. Some straightforward solutions for this problem are described in Section 4.2. A survey of prior dispatching techniques including a detailed description of interval containment is the subject of Section 4.3. Our new slicing technique is described in Section 4.4. The data set of the 35 hierarchies used in our benchmarking, collected from both single and multiple dispatching languages, is presented in Section 4.5. Section 4.6 presents the experimental results, comparing the performance of our algorithm with those of previous algorithms. The application of our results to the problem of multiple dispatching is presented in Section 4.7. An incremental algorithm for constant-time subtyping tests is the concern of Section 4.8. Finally, Section 4.9 mentions open problems and directions for future research. Section 4.10 describes our heuristic for performing type slicing.

4.1 Problem Definitions

Formally, a *hierarchy* is a partially ordered set $(\mathcal{T},\preceq)$ where $\mathcal{T}$ is a set of types and $\preceq$ is a reflexive, transitive and anti-symmetric *subtype relation*. If a and b are types, and $a \preceq b$ holds, we say that a is a *subtype* (or a *descendant*) of b and that b is a *supertype* (or an *ancestor*) of a. Direct subtypes (supertypes) are called *children* (*parents*).

Similarly, we abstract away from the nomenclature of different languages, and use the term *message* for the unique identifier of a family of *methods* (also called member functions, operations, features, implementations, etc.). A message, which is sometimes called a *selector* (in e.g., SMALLTALK [71] or OBJECTIVE-C [36]) or a *signature* (in e.g., JAVA [7] or C++ [124]), may include, depending on the programming language, components such as name, arity, and even the type of parameters. We will use the terms *message* and *selector* interchangeably. Note that a consequence of feature renaming in EIFFEL [97], is that the message does not always include the name of a routine. The intuition however is the same in all OO languages: when an object receives a message encoded as a selector, *dispatching on the type of the receiver* must take place at runtime to find and invoke the implementation which is most appropriate for the receiver's type.

We use the following notation. The $\min$ operator return the set of smallest types in any given set:

$$\min(X) = \{t \in X \mid \nexists t' \in X : t' \neq t, t' \preceq t\}. \tag{4.1}$$

Let $F \subseteq \mathcal{T}$ denote the *family* of types which have a *method implementation* for the same *message*. Given a family F and a type t, $\mathsf{cand}(F, t)$ is the set of candidates in F, i.e., those ancestors of t in which an implementation of the given message exists:

$$\mathsf{cand}(F, t) \equiv F \cap \mathrm{ancestors}(t). \tag{4.2}$$

A *dispatching query* $\mathsf{dispatch}(F, t)$ returns either *the smallest candidate* or null if no such unique candidate exists. (A null result represents either the *message not understood* or *message ambiguous* error conditions.) Specifically,

$$\mathsf{dispatch}(F, t) \equiv \begin{cases} t' & \text{if } \min(\mathsf{cand}(F, t)) = \{t'\}, \\ \mathrm{null} & \text{otherwise.} \end{cases} \tag{4.3}$$

Figure 4.1 depicts a hierarchy which serves as the running example of this chapter. Type names are written with uppercase letters; messages with lower case letters.

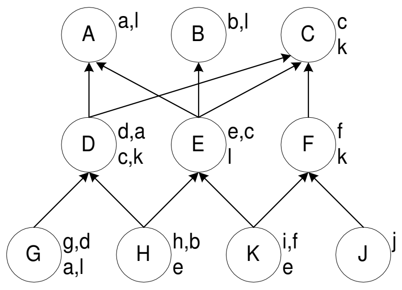


Figure 4.1: A small example of a hierarchy and the methods implemented in each type

For instance, we see in the figure that for the family $F_{\mathsf{c}} = \{\mathsf{C}, \mathsf{D}, \mathsf{E}\}$:

$\mathsf{cand}(F_{\mathsf{c}}, \mathsf{K})$= $\{\mathsf{C}, \mathsf{E}\}$	$\mathsf{dispatch}(F_{\mathsf{c}}, \mathsf{K})$= E
$\mathsf{cand}(F_{\mathsf{c}}, \mathsf{B})$= $\emptyset$	$\mathsf{dispatch}(F_{\mathsf{c}}, \mathsf{B})$= null (*message not understood*)
$\mathsf{cand}(F_{\mathsf{c}}, \mathsf{H})$= $\{\mathsf{C}, \mathsf{D}, \mathsf{E}\}$	$\mathsf{dispatch}(F_{\mathsf{c}}, \mathsf{H})$= null (*message ambiguous*)

The type checker of statically typed languages makes sure at compile time that dispatching never results in null. It would therefore be a compilation error in statically typed language to send c to objects whose static type is B or H. Moreover, it is a compilation error even to send this message to any ancestor of H, e.g., type C. The reason is that the type analyzer cannot infer [66] that the dynamic type is not H.

We shall assume a pre-processing stage in which all ambiguities are resolved by an appropriate augmentation of the families. In the example, we add type H to the family F_{c} since $\mathsf{dispatch}(F_{\mathsf{c}}, \mathsf{H})$ resulted in *message ambiguous*. As in previous work [77, 112] in which this assumption was made, our working hypothesis is that the ensuing increase of problem size is insignificant in practice.

Figure 4.1 is an example of a *multiple inheritance* hierarchy, since, e.g., type D has two parents: A and C. *Single inheritance*, in which each type has at most one parent, is mandated by languages such as SMALLTALK and OBJECTIVE-C. The fact that single inheritance hierarchies take a simple forest topology, makes single inheritance an important special case, for which very efficient algorithms exist. The general case of multiple inheritance is more difficult, and will be our main concern here.

Definition 4.1 *Given a hierarchy ($\mathcal{T}$,$\preceq$) and a family collection $\mathcal{F} \subseteq \wp(\mathcal{T})$, the* dispatching problem *is to* encode *the hierarchy in a data structure supporting* $\mathsf{dispatch}(F, t)$ *queries for all $F \in \mathcal{F}$, $t \in \mathcal{T}$.*

From a practical point of view we assume that each object includes an accessible type-id, and tacitly ignore the object space overheads and the time of retrieving such type-id. Also, the message is given at runtime as an integer selector. We usually assume that this selector is known at compile time, and accordingly allow any pre-processing which is dependent solely on this selector. Given the object type-id and this selector, the dispatch query means that the runtime system must compute the address of the method defined in the smallest candidate, and jump to it.

A solution to the dispatching problem is measured by the following three metrics: (i) the space that the data structure occupies in memory, (ii) the time required for processing a query, and (iii) the time for generating the data structure from the input hierarchy. We would like to express these metrics as a function of the following parameters of the problem:

- The number of types in the hierarchy

$$n \equiv |\mathcal{T}|. \tag{4.4}$$

- The number of different messages that can be sent during runtime

$$m \equiv |\mathcal{F}|. \tag{4.5}$$

- The total number of different method implementations

$$\ell \equiv \sum_{F \in \mathcal{F}} |F|. \tag{4.6}$$

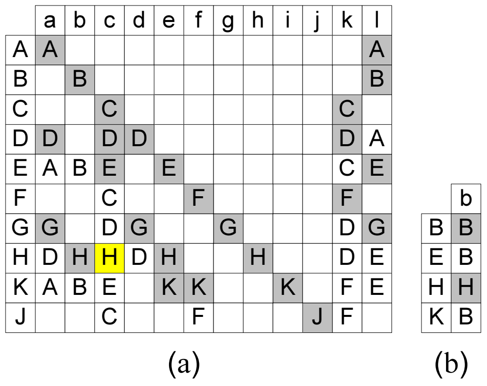
- The number of valid message-type combinations, i.e., combinations which do not result in null

$$w \equiv |\{\langle F, t\rangle \mid \mathsf{dispatch}(F, t) \neq \mathrm{null}\}|. \tag{4.7}$$

4.2 Straightforward Solutions

The most obvious solution to the dispatching problem is in a $n \times m$ *dispatching matrix*, storing the outcomes of all possible dispatching queries. We stress that the order of rows and columns in the dispatching matrix is arbitrary, and the performance of some algorithms for compressing the matrix may depend heavily on the chosen ordering.

The dispatching matrix of our running example is presented in Figure 4.2(a), where the $nm - w$ type-message pairs which result in null are represented as empty entries. The figure depicts in grey all ℓ entries which represent a method implemented in a certain type. For example, the top right grey entry is to say that type A has an implementation of message l. (Recall that type H was added to family F_{c} to resolve an ambiguity. Therefore, the cell corresponding to $\langle \mathsf{H}, \mathsf{c} \rangle$ is rendered in grey.)



	a	b	c	d	e	f	g	h	i	j	k	l
A	A											A
B		B										B
C			C								C	
D	D		D	D							D	A
E	A	B	E		E						C	E
F			C			F					F	
G	G		D	G			G				D	G
H	D	H	H	D	H			H			D	E
K	A	B	E		K	K			K		F	E
J			C			F				J	F	

(a)

	b
B	B
E	B
H	H
K	B

(b)

Figure 4.2: (a) The dispatching matrix, and (b) the sorted dictionary for message b

In the matrix representation queries are answered by a quick indexing operation. However, the space consumption is inhibitivly large, e.g., 512MB for the dispatching matrix in the largest hierarchy in our benchmarks (8,793 types and 14,575 messages).

There are two opportunities for compressing the dispatching matrix:

Null elimination There is much empirical evidence to show that dispatching matrices are very sparse. *Null elimination*, which was the objective of almost all previous work, is the attempt to store only the non-null elements in the matrix.

The ratio $(nm)/w$ is an upper bound on the compression rate which null elimination might achieve. The matrix of Figure 4.2(a) has $120 = 10 \times 12$ entries, out of which, 46 are non-null. Null elimination in this case gives a compression factor of no more than $120/46 \approx 2.6$. In our benchmarks we found that on average, null elimination might achieve compression by a factor of circa 150.

Null elimination can be achieved by storing each column as a *sorted dictionary*, i.e., a sorted array of ⟨key,value⟩-pairs. In the running example, the sorted dictionary for message b is depicted in Figure 4.2(b). In this implementation, the query time is logarithmic in the number of non-null entries in each column. Space is linear in this number.

Dynamic perfect hashing (DPH) [42] is theoretically better than sorted dictionaries. In this algorithm each column (or the entire matrix for that matter) is stored as a hash table. Indices (or their concatenation) serve as keys. The space requirement is linear in w. More importantly, query time is constant! Unfortunately, DPH is of mere theoretical interest since it carries large hidden constants, which might offset any saving of space due to null elimination.

The more sophisticated previously published practical algorithms, try, and in most cases achieve complete, or almost complete null elimination with no hidden constants and constant search time.

Duplicates elimination Even though optimal null elimination may give very good results, it still leaves something to be desired. In one hierarchy of our data set, featuring 3,241 types, an optimal null elimination scheme still requires 2.4MB. *Duplicates elimination* improves on null elimination by attempting to store only the *distinct* ℓ entries of the dispatching matrix. Therefore, the compression factor of duplicates elimination is at most $(nm)/\ell$, which was around 725 in our benchmarks.

The ratio w/ℓ gives the factor by which duplicates elimination can improve on null elimination. This ratio was as high as 122.4 in one of our benchmarks. In the matrix of Figure 4.2(a) there are 27 distinct entries, i.e., $\ell = 27$, so duplicates elimination has the potential of compressing the dispatching matrix by a factor of $120/27 \approx 4.44$.

It is not difficult to come close to full duplicates elimination, with a simple representation of the hierarchy as a graph where types are nodes and immediate inheritance relations are edges. The cost is of course the search time, which becomes $O(n)$, since each dispatch must traverse all the ancestors of a receiver in order to find the smallest candidate. Sophisticated caching algorithms make the typical case more tolerable than what the worst case indicates. This is the implementation in languages such as SMALLTALK.

Our challenge here is to come as close as possible to optimal duplicates elimination, i.e., space linear in the number of implementations ℓ, while still maintaining small, preferably constant, query time.

4.3 Previous Work

This section gives an overview of some of the dispatching techniques proposed in the literature. The performance of these techniques might be improved by using various forms of caching at runtime (see e.g., [31, 37, 78]).

VFT: *Virtual Function Tables* [57] As mentioned above, the VFT technique is valid only in statically typed languages [132]. In a single inheritance setting, VFT achieves optimal null elimination and constant dispatch time. A distinguishing property of the technique is that it does not require whole program information. The VFT of any type can be constructed using only information regarding its ancestors.

The multiple inheritance version of the VFT is much more complicated than the single inheritance version, with complicated space and time overheads. Each type stores multiple VFTs, and if a method is inherited along more than one path, then it will be stored in these more than once. Further, in presence of shared (virtual) inheritance, searching for an implementation is carried out by either following a chain of pointers to ancestors, or by additional increase to object size using *inessential virtual base pointers* [68]. It was shown [53] that these space overheads can be very significant. Even with this overhead, dispatching time increases due to what is known in the C++ jargon as `this`-*adjustment*.[1]

RD: *Row Displacement* [45,47] Another null elimination technique is due to Driesen [45] who suggested to displace the rows in the dispatching matrix by different offsets so that they could be merged together in a *master array*. Later [47] it was found that *selector based RD*, i.e., a displacement of columns rather than rows, gives much better compression values. In fact this technique comes very close (median value 94.7%) to optimal null elimination.

In dynamically typed languages vanilla RD does not work, since null entries which correspond to *message not understood* will usually become occupied. It is possible to amend RD with an increase to space requirement and adding one more comparison at runtime.[2] We stress that duplicates elimination (which we use) does not suffer from this limitation.

CT: *Compact dispatch Tables* [130–132] The very good compression results of RD were improved significantly by Vitek and Horspool on some hierarchies. Their CT technique aims at duplicates elimination. The idea is to partition the set of families $\mathcal{F}$ into disjoint *slices* $\mathcal{F}_1, \ldots, \mathcal{F}_k$. Slicing breaks the dispatching matrix into k sub-matrices, also called *chunks*. Identical rows within each chunk are then merged. Each type t has an array r_t of size k. Entry $r_t[i]$ points to the row of t in chunk i. Dispatching in CT requires an extra load compared to the dispatching matrix, but the merging of rows may reduce the space requirement.

Our algorithms for multiple inheritance adopt the slicing idea. However, we slice the set of types rather than the set of families.

SC: *Selector Coloring* [44, 118] SC aims at null elimination by slicing the set of messages. Each slice must satisfy the following property: *no two messages in the slice can be recognized by the same type.* In other words, in each chunk, a row can have at most one non-null entry. This property makes it possible to merge together all the columns in a chunk, resulting in a space requirement of $n \times k$.

The performance of SC is improved as the number of slices decreases. Since it is computationally hard to find an optimal slicing, the slices must be found using a heuristic.

[1]In general, dispatching in C++ is tightly coupled with its peculiar object-layout, and is therefore not directly applicable to languages with different layout scheme. Simple object-layout have the advantage of fast synchronization, hash-codes, and easier garbage collection.

[2]The trick is to add a prologue to each method which checks that the method indeed corresponds to the sent message.

As in RD, null entries are treated as empty in SC and therefore additional storage and an extra comparison are required in dynamically typed languages. CT also uses SC in each of the chunks.

***Jalapeño* [3]** JAVA's `invokeinterface` bytecode instruction, i.e., messages sent to receivers whose static type is an **`interface`**, cannot be implemented using the VFT technique. Jalapeño, an IBM implementation of JAVA virtual machine, uses a fast incremental variant of SC in realizing these instructions. Messages are hashed into k slices, where k is an a-priori fixed number. Each type has an *interface method table* of length k. When the slicing property of SC does not hold, i.e., some type recognizes more than one message in the same slice, then a conflict resolution thunk must be generated by the compiler. Since there is no bound on the number of conflicting messages in each hash table entry, dispatch time is not necessarily constant. It is easy to see that the total memory requirement is nk for the tables, plus $O(w)$ memory for conflict resolution.

***Interval Containment for single inheritance hierarchies* [60,100]** Interval containment achieves optimal duplicates elimination at the cost of non-constant dispatch time. Our dispatching technique is a generalization of interval containment for multiple inheritance hierarchies. Let us describe this technique in greater detail.

Interval containment assigns id's to types in a preorder traversal of the tree hierarchy. An important property of the preorder traversal is that descendants of a type t define an *interval*. Therefore, each family F, defines a set of intervals, one for each type $t \in F$.

Figure 4.3(a) shows a tree hierarchy with three implementations of a message a in types: A, B, and F, i.e., family $F_{\mathsf{a}} = \{\mathsf{A}, \mathsf{B}, \mathsf{F}\}$. Then, as can be seen in Figure 4.3(b), these family members define three intervals in the preorder traversal: $[1, 7]$, $[5, 7]$, and $[3, 3]$, respectively. The intersections of those three intervals partition the types into four segments: $[1, 2]$, $[3, 3]$, $[4, 4]$, and $[5, 7]$, which correspond to family members: A, F, A, and B, respectively. The dispatch of message a on any given type depends only on the segment this type belongs to. If, for example, the receiver is of type G whose id is 6, then we find that it belongs to segment $[5, 7]$, and therefore return B.

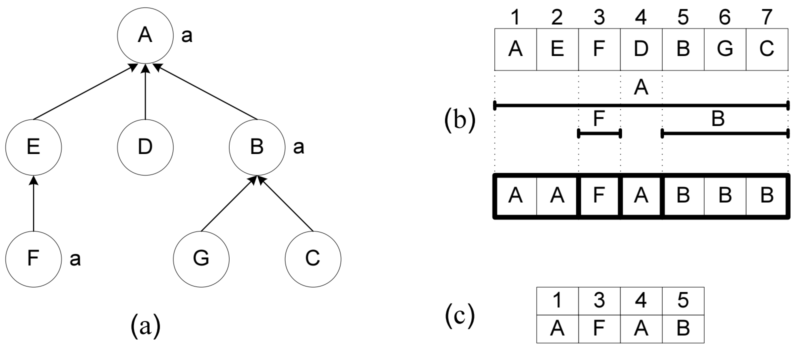


Figure 4.3: (a) A family $F_{\mathsf{a}} = \{\mathsf{A}, \mathsf{B}, \mathsf{F}\}$ in a tree hierarchy, (b) the intervals and segments F_{a} defines, and (c) the representation of F_{a} as a sorted dictionary

Given a family F, there are $|F|$ intervals which partition $\mathcal{T}$ into at most $2|F| + 1$

segments, where all types in a segment have the same dispatching result. Family F is represented as a sorted dictionary, mapping segments' starting point to methods. In our example, Figure 4.3(c) shows a sorted dictionary that represents the segment partitioning. This dictionary serves as the dispatching table for F_{a}.

Note that the sorted dictionary representation is linear in $|F|$. The total memory for representing all families is therefore $O(\ell)$. In fact, the number of memory cells required by this representation is at most

$$\sum_{F\in\mathcal{F}} 2(2|F|+1) = 2m + 4\sum_{F\in\mathcal{F}}|F| = 2m + 4\ell.$$

It remains to describe the representation of the sorted dictionary and the procedure to determine the segment to which a specific type belongs. Algorithmically, the problem is characterized as follows: Given a set of integers $S \subseteq [1,\ldots,n]$, build a data structure to implement the predecessor operation, $\mathrm{pred}(x)$, defined as

$$\mathrm{pred}(x) = \max\{y \in S \mid y \leq x\}, \tag{4.8}$$

for any integer $x \in [1,\ldots,n]$. Let $s = |S|$. In our case, s, which is smaller than twice the number of different implementations, is typically much smaller than n. We will therefore be more interested by algorithms whose resource demands are dependent on s, rather than on n.

In an array implementation it is possible to implement $\mathrm{pred}(x)$ using a *binary search* in $O(\log s)$ time, while the space requirement is $O(s)$. The hidden constants are small.

If the number of integers is not so small, then a theoretically superior algorithm is the *Q-fast trie* [134], which achieves $O(\sqrt{\log n})$ time while still maintaining the space linear in s. Stratified trees, also called *van Emde Boas data structure* [128, 129], offer a different tradeoff, with space linear in n and time $O(\log\log n)$. In the randomized version of stratified trees the expected space requirement is reduced to $O(s)$. In practice we expect the simple binary search algorithm to outperform these asymptotically better competitors.

4.4 Dispatching using Type Slicing

Our dispatching technique for multiple inheritance hierarchies is a generalization of *interval containment* for single inheritance hierarchies. The idea behind interval containment is that there is an ordering of the tree hierarchy in which the descendants of any given type are consecutive. The difficulty in the multiple inheritance case is that an ordering of $\mathcal{T}$ with the above property might not exist. Figure 4.4 shows the smallest hierarchy for which such an ordering is impossible. The reason is that such an ordering imposes the contradicting constraints that A, B and C must be adjacent to D.

Instead of imposing a global ordering, we partition the set of types $\mathcal{T}$ into disjoint *slices* $\mathcal{T}_1,\ldots,\mathcal{T}_\kappa$ and impose a local ordering condition on each of the slices. For a slice $\mathcal{T}_i$ and a type t (not necessarily in $\mathcal{T}_i$), let $D_i(t)$ be the set of descendants of t in $\mathcal{T}_i$,

$$\begin{tikzcd}
A & B & C \\
& D \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Figure 4.4: The smallest multiple inheritance hierarchy for which no ordering exists where all descendants of any type are consecutive

i.e.,

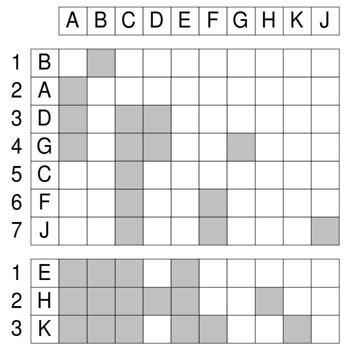
$$D_i(t) = \mathrm{descendants}(t) \cap \mathcal{T}_i\,.$$

Figure 4.5 shows a partitioning of the hierarchy of Figure 4.1 into two slices:

$$\mathcal{T}_1 = \{\mathsf{B}, \mathsf{A}, \mathsf{D}, \mathsf{G}, \mathsf{C}, \mathsf{F}, \mathsf{J}\},$$
$$\mathcal{T}_2 = \{\mathsf{E}, \mathsf{H}, \mathsf{K}\}.$$

The grey entries in any column represent a set of descendants of some type. The sets of descendants of type A, for example, in the two slices are

$$\begin{aligned} D_1(\mathsf{A}) &= \{\mathsf{A}, \mathsf{D}, \mathsf{G}\}, \\ D_2(\mathsf{A}) &= \{\mathsf{E}, \mathsf{H}, \mathsf{K}\}. \end{aligned} \tag{4.9}$$



		A	B	C	D	E	F	G	H	K	J
1	B		▨								
2	A	▨									
3	D	▨		▨	▨						
4	G	▨		▨	▨			▨			
5	C			▨							
6	F			▨			▨				
7	J			▨			▨				▨
1	E	▨	▨	▨		▨					
2	H	▨	▨	▨	▨	▨			▨		
3	K	▨	▨	▨		▨	▨			▨	

Figure 4.5: Type slicing for the hierarchy of Figure 4.1

The type slicing technique is based on the demand that the sets $D_i(t)$ are *consecutive* in some ordering of the rows. Visually this means that the grey entries are consecutive within each chunk. For instance, in Figure 4.5 the sets of (4.9) define the *intervals*

$$\begin{aligned} D_1(\mathsf{A}) &= [2, 4], \\ D_2(\mathsf{A}) &= [1, 3]. \end{aligned} \tag{4.10}$$

Formally, each slice $\mathcal{T}_i$ must satisfy the following slicing property:

There is an ordering of $\mathcal{T}_i$ in which $D_i(t)$ is consecutive for all types $t \in \mathcal{T}$.

Each type t is identified by a pair $\langle s_t, \mathrm{id}_t \rangle$, where s_t is an id of the slice to which t belongs, and id_t is the position of t in the ordering of this slice. Thanks to the slicing property, the set $D_i(t)$ defines an *interval* for each i, $1 \le i \le \kappa$.

A partitioning of $\mathcal{T}$ into slices which satisfy the slicing property always exists, since this property trivially hold for singletons. We will strive to minimize κ, the total number of slices.

Finding the slices We are unaware of any non-exponential method for finding the minimal number of slices. Instead we use a greedy heuristic: "try to make the current slice as large as possible without violating the slicing property". Specifically, we traverse the types in a topological order, and try to insert each type into each of the slices. If all these insertion attempts fail then a new slice is created.

Given a slice $\mathcal{T}_i$ and a type t, PQ-trees [17, 136] can be used to check whether there is *any* ordering of $\mathcal{T}_i \cup \{t\}$ which satisfies the slicing property, in $O(n \cdot |\mathcal{T}_i|)$ time. In inserting n types using this strategy, the total time might be cubic in n, which is highly undesirable.

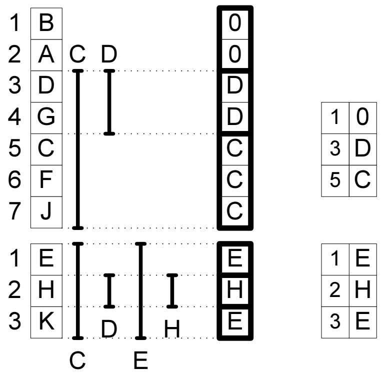
Instead we use a heuristic which, by not disturbing the existing order of $\mathcal{T}_i$, achieves a run time that depends only on the number of ancestors of t. Therefore, the total runtime of the above algorithm for finding the slices is $O(\kappa|\preceq|)$. The exact details of the heuristic are presented in Section 4.10.

Dispatching using type slicing Given a type t and a family F, a dispatching query returns the smallest type $t' \in F$ such that $t' \succeq t$. Let $\mathcal{T}_i$ be the slice of t. Given a type t', we have that $t' \succeq t$ iff $t \in D_i(t')$. We therefore must consider all intervals of $D_i(t')$, $D_i(t') \neq \emptyset$, where $t' \in F$. Since there are at most $|F|$ such intervals, we obtain a partition of $\mathcal{T}_i$ into $2|F| + 1$ segments, where the result of the dispatch on t depends only on the segment to which t belongs.

Figure 4.6 shows the dispatching representation for the family

$$F_{\mathsf{c}} = \{\mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{H}\}$$

in the hierarchy of Figure 4.1. Consider, for example, the first slice. Only types C and D define non empty intervals, which are $[3, 7]$ and $[3, 4]$, respectively. We also consider the implicit interval $[1, 7]$ for the method *message not understood*. Those three intervals partition the types into three segments: $[1, 2]$, $[3, 4]$, and $[5, 7]$. Message c is represented in the first slice using an appropriate data structure storing those three segments, and mapping them to: null (*message not understood*), D, and C, respectively.



(a) (b)

Figure 4.6: (a) The intervals and segments of message c in the two slices of Figure 4.5, and (b) the message representation in each slice

In general, a family F is encoded in slice $\mathcal{T}_i$ by a data structure of choice which represents a set of segments, mapping each one to the appropriate method implementation. As in vanilla interval containment, this data structure can be a simple array, a Q-fast trie, or a stratified tree. Obviously, each slice has its own unique such data structure.

Dispatching on type $t \in \mathcal{T}$ and family $F \in \mathcal{F}$ is carried out in three stages:

1. Finding s_t, the id of the slice of t,
2. following this slice to find the respective data structure of F, and then
3. carrying on as in single inheritance in a search of id_t in this data structure to find the dispatching result.

Thus, dispatching in multiple inheritance hierarchies requires only two more steps in comparison to dispatching in single inheritance hierarchies. The space requirement in multiple inheritance hierarchies increases by a factor of at most κ. Curiously, this factor depends only on the topology of the hierarchy and the quality of the slicing algorithm. It does not depend in any way on the number of messages.

Reducing the number of slices We now describe one optimization that given the set of messages reduces the number of slices κ. In our multiple inheritance benchmarks, κ is reduced by an average of 1.35. (In the LOV hierarchy, for example, the number of slices is reduced from 12 to 7.) The key observation is that the dispatching algorithm assumes that each family member $t \in F$ defined an interval for each slice. Therefore, $D_i(t)$ must be consecutive in $\mathcal{T}_i$, *only* for those types t which are indeed members in some family F.

Formally, we say that a type t is *significant* if there exists a family F such that $t \in F$, and redefine the slicing property as follows:

> *There is an ordering of $\mathcal{T}_i$ in which $D_i(t)$ is consecutive for all significant types $t \in \mathcal{T}$.*

Optimizations for statically typed languages We also note that in *statically typed languages*, the *binary search* algorithm can be optimized. Suppose that we dispatch on an object whose *static type* is a. Then, at runtime, the binary search can begin at a smaller interval, restricted only to the interval of descendants of a in each of the slices.

Moreover, we can even discard segments which correspond to *message not understood*, since such a case does not occur in statically typed languages.

4.5 Data Set

The data set for benchmarking dispatching algorithms has 16 single inheritance hierarchies with 29,162 types, 12 multiple inheritance hierarchies with 27,728 types, and seven multiple dispatch hierarchies with 7,082 types.

This benchmark includes 5 hierarchies out of 13 hierarchies used in previous experimental work on subtyping. (We were unable to obtain information on the definition of messages and methods in the other eight hierarchies.) As observed previously [53] many of the topological properties of these hierarchies are similar to those of balanced binary trees. The average number of ancestors in these hierarchies is less than 9 for all hierarchies, with the exception of Geode, in which it is 14.0 and Self, in which it is 30.9.

All degenerate families, i.e., families of size one (singletons), were eliminated from the data set prior to running the experiments, since no runtime dispatching is required for such families.

We stress that by eliminating degenerate families we only made the input *more difficult* for our new dispatching algorithm and any other duplicates elimination scheme, including CT. The reason is that degenerate families, in which there are only two distinct values in their corresponding columns, have the greatest potential for duplicates elimination.

Table 4.1 gives a summary of the pruned hierarchies. The three blocks in the table correspond to single inheritance-, multiple inheritance-, and multiple dispatch- hierarchies. We see that the hierarchies span a range of sizes, from about a hundred types up to almost 9,000 types.

The row denoted *Total* in this and some of the subsequent tables corresponds to the total or universal hierarchy obtained by a simple disjoint union of all hierarchies in the ensemble. In most cases, the "Total" row therefore corresponds to an average of the different hierarchies, weighted by size. In Table 4.1, this row indicates that in total the dispatching benchmark spanned some 64 thousand types and 70 thousand messages.

The ℓ/n column shows the average number of method implementations per type. Examining the entries along this column we see that in many multiple dispatch hierarchies, there are about one or two methods per type. A typical value of the other hierarchies is four or five implementations per type. The San Francisco (SI: IBM SF) project gives the largest number of methods per type (13.3).

In checking the ℓ/m column we find that families tend to be small, with average values of around four to six methods in a family in most hierarchies. We note that the average number of comparisons in a binary search in families is no greater than $\lceil \log_2 \frac{\ell}{m} \rceil$. The reason is that the geometrical mean is no greater than the arithmetical mean, and therefore

$$\begin{aligned} \frac{1}{m}\sum_{F\in\mathcal{F}} \log_2|F| &= \log_2 \left(\prod_{F\in\mathcal{F}}|F|\right)^{\frac{1}{m}} \\ &\le \log_2 \left(\frac{1}{m}\sum_{F\in\mathcal{F}}|F|\right) \\ &= \log_2 \frac{\ell}{m}. \end{aligned} \tag{4.11}$$

Thus, just by inspecting the ℓ/m column we learn that the number of comparisons is about 3.

The next $(nm)/w$ column gives the best possible factor by which null elimination can improve upon the complete dispatching matrix. As can be seen from the table, this matrix

	Hierarchy	n	m	ℓ/n	ℓ/m	$(nm)/w$	w/ℓ
Single Inheritance	Visualworks1	774	1,170	6.0	4.0	11.4	17.1
	Visualworks2	1,956	3,196	6.9	4.2	21.6	21.3
	Digitalk2	535	962	6.2	3.5	7.1	21.7
	Digitalk3	1,357	2,402	7.0	3.9	9.0	38.3
	IBM Smalltalk 2	2,320	4,335	7.0	3.8	49.1	12.6
	VisualAge 2	3,241	6,529	8.1	4.0	35.6	22.7
	NextStep	311	499	6.8	4.2	9.6	7.7
	ET++	371	296	3.8	4.8	9.0	8.6
	SI: JDK 1.3.1	6,681	4,392	3.6	5.4	228.8	5.4
	SI: Corba	1,329	222	1.9	11.6	42.5	2.7
	SI: HotJava	644	690	4.5	4.2	18.6	8.2
	SI: IBM SF	6,626	11,664	13.3	7.6	268.9	3.3
	SI: IBM XML	107	131	5.5	4.5	10.8	2.2
	SI: Orbacus	1,053	980	3.6	3.9	55.3	4.9
	SI: Orbacus Test	579	368	4.1	6.5	37.6	2.4
	SI: Orbix	1,278	535	2.3	5.4	62.7	3.8
Multiple Inheritance	Self	1,802	2,459	12.1	8.8	18.9	10.8
	Unidraw	614	360	3.8	6.5	27.3	3.5
	LOV	436	663	6.5	4.3	20.5	5.0
	Geode	1,318	1,413	7.2	6.7	15.2	12.9
	MI: JDK 1.3.1	7,401	5,724	3.9	5.0	300.7	4.9
	MI: Corba	1,699	396	1.9	8.1	49.6	4.2
	MI: HotJava	736	829	4.6	4.1	24.5	7.3
	MI: IBM SF	8,793	14,575	13.2	8.0	328.3	3.4
	MI: IBM XML	145	271	6.5	3.5	16.9	2.5
	MI: Orbacus	1,379	1,261	3.6	4.0	70.1	5.0
	MI: Orbacus Test	689	379	4.0	7.3	34.9	2.7
	MI: Orbix	2,716	786	1.4	4.7	95.1	6.1
Multiple Dispatching	Cecil	932	1,009	4.5	4.2	12.9	17.3
	Dylan	925	428	1.9	4.2	5.6	39.5
	Cecil-	473	592	5.0	4.0	17.4	6.8
	Cecil2	472	131	1.2	4.3	3.6	30.6
	Harlequin	666	229	1.5	4.4	6.6	22.7
	Vor3	1,660	328	1.1	5.7	35.3	8.3
	Vortex3	1,954	476	1.3	5.2	3.0	122.4
	Total	63,972	70,680	6.5	5.9	1,242.0	8.7
	Median	1,053.0	690.0	4.5	4.4	21.6	7.3
	Minimum	107	131	1.1	3.5	3.0	2.2
	Maximum	8,793	14,575	13.3	11.6	328.3	122.4

Table 4.1: Statistical and topological properties of the 35 hierarchies used in benchmarking dispatching algorithms

is very sparse. In most cases, 90% or more of its cells are null. In hierarchies such as MI: JDK 1.3.1 and MI: IBM SF we even find that the potential compression is by a factor as high as 300. (The 1,242.0 bound for the universal hierarchy is meaningless.)

How much can duplicates elimination improve on an *optimal* null elimination? The answer is in the w/ℓ column. We observe a potential for *additional* compression by factors of about 10. Duplicates elimination performs very well precisely on the multiple dispatch hierarchies, where mere null elimination is not as effective as it is in other hierarchies.

4.6 Experimental Results

In order to evaluate the quality of the order-preserving heuristic used in our TS technique, we compared it with a much more powerful, but time consuming, heuristic which uses PQ-trees. The superscript PQ shall denote the variant which use the PQ heuristic.

Space requirement We follow the popular convention of ignoring *code space* requirement, i.e., assuming that there is a single generic dispatching routine which receives a message-selector and a type-id. Although our results indicate that inlining of the binary search might be worthy, further research is required to estimate the incurred code space penalty. The following definition is pertinent to the comparison of algorithms.

Definition 4.2 *Let W be the number of 4-bytes words the algorithm uses to encode the dispatching tables of a certain hierarchy, then the algorithm's* redundancy factor *on this hierarchy is W/ℓ.*

In other words, the redundancy factor of a dispatching algorithm in a certain hierarchy is the ratio between the total space requirement of that algorithm and the lower bound ideal implementation which uses 4 bytes for storing the address of each method.

Table 4.2 gives the redundancy factor of different algorithms on the 35 hierarchies in our dispatching benchmark. In reading the table, remember that better algorithms have lower redundancy factors.

Algorithms CT, TS, and TS$^{\text{PQ}}$ attempt to achieve duplicates elimination. The other algorithms rely on null elimination. The results in the table do not include the additional provisions mentioned above for the RD, CT, and SC algorithms to support dynamically typed languages. The redundancy factors have to be appropriately adjusted to include selector verification information.

Since we did not have access to the original implementation and heuristics of SC and CT, redundancy factors reported in the respective columns present a lower bounds on these values: In SC, the number of slices is no less than the maximal number of messages that a type understands. In estimating CT, the set of messages was divided into chunks of 14 messages each (as prescribed in [132]). We then applied the SC lower bound estimate in each chunk.

The results of the VFT technique are calculated in the traditional manner [47], under the assumption that there are no virtual bases. The size of a type VFTs equals the sum

	Hierarchy	CT	VFT	SC[a]	RD	**TS^{PQ}**	**TS**	Mem[b]
Single Inheritance	Visualworks1	18.3	17.1	24.3	17.3	2.8	2.5	45
	Visualworks2	37.5	21.3	39.8	21.7	2.6	2.5	134
	Digitalk2	15.8	21.7	59.8	22.0	3.0	2.7	35
	Digitalk3	29.8	38.3	92.5	38.8	3.0	2.7	98
	IBM Smalltalk 2	48.9	12.6	37.5	15.4	3.0	2.6	165
	VisualAge 2	63.0	22.7	62.3	29.2	3.0	2.6	267
	NextStep	10.7	7.7	21.8	7.9	2.9	2.6	22
	ET++	9.9	8.6	26.0	8.9	2.6	2.4	13
	SI: JDK 1.3.1	91.9	5.4	67.9	6.2	2.6	2.4	219
	SI: Corba	10.1	2.7	25.2	3.7	2.8	2.7	27
	SI: HotJava	15.5	8.2	33.7	8.5	2.8	2.5	28
	SI: IBM SF	66.0	3.3	26.0	3.5	2.4	2.2	744
	SI: IBM XML	4.2	2.2	8.4	2.5	2.5	2.1	5
	SI: Orbacus	22.6	4.9	35.0	5.1	2.8	2.4	36
	SI: Orbacus Test	8.4	2.4	43.9	2.9	2.5	2.3	21
	SI: Orbix	21.3	3.8	35.7	4.6	2.8	2.5	29
Multiple Inheritance	Self	17.6	10.8	27.3	11.1	3.0	2.8	240
	Unidraw	10.7	3.5	15.3	4.0	2.7	2.5	23
	LOV	12.1	12.8	11.8	5.2	4.4	4.5	50
	Geode	19.2	44.9	40.4	16.2	5.5	6.1	228
	MI: JDK 1.3.1	109.2	5.8	62.4	5.5	4.1	4.1	463
	MI: Corba	18.5	6.5	35.6	4.9	3.4	3.3	42
	MI: HotJava	17.3	8.5	39.0	7.6	4.2	4.6	60
	MI: IBM SF	82.3	5.9	26.2	3.5	3.8	3.7	1,663
	MI: IBM XML	5.7	3.5	8.7	2.6	3.5	3.3	12
	MI: Orbacus	28.0	6.9	37.5	5.3	4.0	3.8	75
	MI: Orbacus Test	8.8	3.5	45.3	3.0	3.2	3.2	35
	MI: Orbix	45.1	7.0	64.5	6.7	3.6	3.4	49
Multiple Dispatching	Cecil	19.5	34.0	34.6	17.8	4.2	4.1	68
	Dylan	20.5	46.3	71.6	40.2	3.5	3.5	24
	Cecil-	12.7	12.7	27.7	7.2	4.5	4.8	45
	Cecil2	11.6	100.3	69.7	31.2	3.3	3.9	9
	Harlequin	14.2	47.9	83.3	23.5	4.3	4.4	18
	Vor3	24.1	19.4	50.8	9.3	3.4	3.5	26
	Vortex3	29.2	375.7	159.7	124.0	3.5	4.1	40
	Total	55.7	22.8	48.5	13.3	3.3	3.2	433
	Median	18.5	8.5	37.5	7.6	3.0	2.8	42
	Minimum	4.2	2.2	8.4	2.5	2.4	2.1	5
	Maximum	109.2	375.7	159.7	124.0	5.5	6.1	1,663

[a]A lower bound on SC redundancy factor

[b]The space requirements of TS in kilo-bytes

Table 4.2: The redundancy factor of different dispatching algorithms and the total memory requirements of TS in kilo-bytes

of its parents VFTs plus the number of newly introduced messages. However, in practice inheritance is usually *shared* (not *repeated*), giving rise to other overheads [53].

In studying the last column of the table (labeled "Mem") we see that the total space requirement of type slicing ranges between 5KB to almost 1.7MB. When viewed in relative- rather than absolute-terms (in the penultimate column labeled TS showing redundancy factors), we find that the space requirement of type slicing is about three or four times larger than a theoretic optimal duplicates elimination.

In comparing the columns TS and TS^{PQ} we find that using the PQ-heuristic does not always improve the space performance. In fact, in all single inheritance hierarchies, and several multiple inheritance hierarchies, it *increases* the memory consumption of the algorithm. The improvement, in the few cases it occurs, is quite small; a maximum of 15% in the Vortex3 hierarchy.

RD is better than our main TS algorithm in three out of 35 hierarchies: IBM SF (redundancy factor 3.5 in RD vs. 3.7 in TS), IBM XML (2.6 vs. 3.3), and Orbacus Test (3.0 vs. 3.2) multiple inheritance hierarchies. We see that even in these cases the space requirement of TS is comparable to that of RD.

TS however always wins against CT, VFT, SC, and against RD in all other hierarchies, sometimes by factors as large as 30. For instance, in the Vortex3 hierarchy, RD uses 1.24MB, an optimal null elimination scheme will use 1.22MB, while TS uses 40KB!

The average improvement of TS over RD is by a factor of 4.6, while the median improvement is by a factor of 2.6. In fairness, it should be said that all these algorithms dispatch in constant time, using simple array references, while TS uses a non-constant time binary search. This constant time must be extended to include selector verification in dynamically languages, which is not required in TS. Conversely, as we saw in Section 4.4, the search time in TS can be reduced in statically typed languages.

In general, the VFT algorithm is the next best algorithm among *single inheritance hierarchies*. The RD algorithm is usually the second best for *multiple inheritance hierarchies*, while CT performs well on *multiple dispatch hierarchies*.

We remind the reader that the comparison presented in Table 4.2 is different than that reported in the literature, since even though we used the same hierarchies, we eliminated degenerate families from the benchmark. Different algorithms compress such families to different levels.

Creation time Table 4.3 compares the times for creating the compressed dispatching data structures using RD with those of TS and those of TS^{PQ}. Since we could not obtain the original implementations of SC and CT, their runtime is not reported. Vitek and Horspool [132] report that CT required 1.5 seconds for NextStep hierarchy, and 4.8 seconds for Visualworks2, on a Sparc station 5. The implementation of VFT is so straightforward and fast that its runtime overhead can be considered as zero for many practical purposes.

TS is consistently better than RD, sometimes by a factor of hundreds. The average improvement of TS over RD is by a factor of 37.4, while the median is 6.3. (Since RD is a heuristic it may sometimes find a good solution quickly.) TS^{PQ} is very slow.

	Hierarchy	RD	**TS**	**TS**$^{\mathbf{PQ}}$
Single Inheritance	Visualworks1	54	5	261
	Visualworks2	250	13	2,430
	Digitalk2	54	3	130
	Digitalk3	281	9	1,040
	IBM Smalltalk 2	3,430	15	3,790
	VisualAge 2	18,800	24	8,160
	NextStep	13	1	50
	ET++	9	1	60
	SI: JDK 1.3.1	162	26	33,600
	SI: Corba	11	3	561
	SI: HotJava	22	2	211
	SI: IBM SF	1,620	69	30,300
	SI: IBM XML	1	1	10
	SI: Orbacus	27	4	401
	SI: Orbacus Test	12	1	110
	SI: Orbix	18	3	571
Multiple Inheritance	Self	242	30	27,600
	Unidraw	9	3	371
	LOV	18	5	3,430
	Geode	182	38	66,800
	MI: JDK 1.3.1	240	88	324,000
	MI: Corba	26	9	10,400
	MI: HotJava	30	7	3,390
	MI: IBM SF	903	307	1,740,000
	MI: IBM XML	2	1	140
	MI: Orbacus	31	11	12,700
	MI: Orbacus Test	11	4	1,740
	MI: Orbix	31	14	12,400
Multiple Dispatching	Cecil	57	9	6,410
	Dylan	48	5	1,870
	Cecil-	18	4	2,490
	Cecil2	16	1	2,650
	Harlequin	23	2	2,710
	Vor3	24	9	23,400
	Vortex3	394	11	42,100

Table 4.3: Encoding creation time in milliseconds, on a 900 Mhz Pentium III, of different dispatching algorithms

Dispatch time Recall that in TS we associate with each message an array of the κ addresses of the appropriate binary search code in each slice. The main performance metric of such code is the number of conditionals.

We computed the average number of such conditionals, taking care to weigh each slice proportionally to the number of types in it. The average number of such conditionals in the 35 hierarchies ranged between 0.6 and 3.4; the median value being 2.5. (Even though the experiments used only non-degenerate families, i.e., families with two or more methods, it turned out to be that the number of conditionals was sometime zero, precisely when there was only one method implementation in a slice.)

A potentially better technique eliminates the jump by coalescing the jump and the binary branch code of each message. Observe that with this technique dispatching time increases from $O(\log|F|)$ to $O(\log \kappa|F|)$. In practice, if this is implemented then the average number of comparisons ranges between 2.5 to 3.8; the median becomes 2.9. We see that the indirect jump is substituted by about one or two comparisons on average. We should also say that this coalescing technique reduces the total memory requirement, since it eliminates the array of the κ addresses which was associated with each message. We finally note that, for this technique, we can use a weaker definition for the complexity of an hierarchy, which is: *there exists an ordering of $\mathcal{T}$ in which the descendants of any type define at most κ intervals*.

4.7 Multiple Dispatching

Interestingly, our results have applications also to the more general multiple dispatching problem.

4.7.1 Introduction to Multiple Dispatching

Remember that in ordinary dispatching, the method to be invoked depends only on the type of a single receiver. In contrast to this single dispatching, *multiple dispatching* is the dispatch over several arguments. Consider, for example, a geometric modeling application, in which shapes such as rectangles, triangles, circles, are to be depicted on various drawing canvases, such as screens, printers and files. Then, the appropriate drawing method is to be selected according to both the shape and canvas kind. Languages such as POLYGLOT [2], KEA [99], COMMONLOOPS [16], CLOS [15], CECIL [26], DYLAN [120] make only a partial list of the new generation OO languages which support multiple dispatching in the form of *multi-methods*.

Even though multi-methods are believed to be more expressive, natural and readable than *mono-methods*, they did not find their way into more mainstream languages. One of the reasons is probably the perceived cost of implementation. The prospect of efficient multiple dispatching drew much research effort [5, 27, 28, 51, 52, 61, 77, 87, 112]. The contribution that this paper makes is improving the memory requirements of two existing practical techniques of multiple dispatching.

Multiple dispatching can be viewed as dispatching over tuples. Given a hierarchy $(\mathcal{T}, \preceq)$, we define the c*-tuple* hierarchy $(\mathcal{T}^c, \preceq')$, where

$$(a_1, \ldots, a_c) \preceq' (b_1, \ldots, b_c) \text{ iff } \forall i = 1, \ldots, c : a_i \preceq b_i.$$

A multi-method $m(t_1, \ldots, t_c)$ can be thought of as a mono-method defined in the *multi-type* $(t_1, \ldots, t_c) \in \mathcal{T}^c$. However, this perspective does not lead to any efficient algorithms because of the size of the c*-tuple* hierarchy.

4.7.2 Review of Algorithms for Multiple Dispatching

The best practical techniques for multiple dispatching known today are *Compressed N-dimensional Tables* (CNT) [5, 52, 87] and *Single-Receiver Projections* (SRP) [77]. Both techniques begin with the same *mono-dispatch stage*, in which c independent single-dispatch queries are executed for a multi-method of arity c. The results of these queries are then used in the *resolution stage* which is technique specific.

The mono-dispatch stage quickly reduces the number of candidate methods using the following observation. For a given multi-family F, let $T_i(F)$ be the set of all types which occur in position i in some tuple of the multi-family F, i.e., if $(t_1, \ldots, t_i, \ldots, t_c) \in F$ then $t_i \in T_i(F)$. Then, the dispatching of F on a multi-type $(t_1, \ldots, t_c)$, can be made easier by first using a single-dispatch algorithm for finding for each $i = 1, \ldots, c$, the dispatching result $t'_i = \mathsf{dispatch}(T_i(F), t_i)$. Notice that it might be the case that $(t'_1, \ldots, t'_c) \notin F$.

Consider, for example, the multi-family

$$F = \{(\mathsf{A}, \mathsf{A}), (\mathsf{A}, \mathsf{D}), (\mathsf{B}, \mathsf{D}), (\mathsf{E}, \mathsf{D})\}, \tag{4.12}$$

defined over the type hierarchy of our running example (Figure 4.1). Then,

$$\begin{aligned} T_1(F) &= \{\mathsf{A}, \mathsf{B}, \mathsf{E}\}, \\ T_2(F) &= \{\mathsf{A}, \mathsf{D}\}. \end{aligned} \tag{4.13}$$

In dispatching the multi-type (H, D), the mono-dispatch stage first determines that $t'_1 = \mathsf{E}$ and that $t'_2 = \mathsf{A}$. The resolution stage then continues with the multi-type (E, A). Note that even though $(\mathsf{E}, \mathsf{A}) \notin F$, we still have that $\mathsf{dispatch}(F, \mathsf{H}, \mathsf{D}) = \mathsf{dispatch}(F, \mathsf{E}, \mathsf{A})$.

Fact 4.3 (DUJARDIN ET AL. [52, P. 129]). *If dispatching never result in* null *then there is always a unique such t'_i. Further, dispatching on $(t_1, \ldots, t_c)$ is the same as dispatching on $(t'_1, \ldots, t'_c)$, i.e.,*

$$\mathsf{dispatch}(F, t_1, \ldots, t_c) = \mathsf{dispatch}(F, t'_1, \ldots, t'_c).$$

The CNT technique creates a c-dimensional dispatch table with entries for each multi-type in the cartesian product

$$T_1(F) \times \cdots \times T_c(F).$$

The dispatching table for the multi-family of (4.12) is shown in Table 4.4.

	A	D
A	(A, A)	(A, D)
B	null	(B, D)
E	(A, A)	(E, D)

Table 4.4: CNT representation for the multi-family of (4.12)

The resolution stage in CNT requires only $O(c)$ time. The number of memory cells for representing the multi-dimensional dispatch table is reduced from $O(n^c)$ to

$$|T_1(F)| \times \cdots \times |T_c(F)| = O(|F|^c), \tag{4.14}$$

which might still be very large.

SRP gives a different tradeoff in which the time of resolution increases to $O(c|F|)$, while the space (in bits) is

$$|F|\left(|T_1(F)| + \cdots + |T_c(F)|\right) = O(c(|F|)^2). \tag{4.15}$$

An asymptotic comparison of the bound (4.15) with $O((|F|)^c \log|F|)$, the bound on number of bits in the CNT representation obtained from (4.14), as well as practical experience, shows that SRP is usually more space efficient than CNT.

SRP uses an encoding of subsets of F as bit vectors of length $|F|$. The positions in this bit vector are given in a topological order, so that smaller multi-types are positioned first. For all $i = 1, \ldots, c$, and for all $t \in T_i(F)$, the technique encodes the set of all family members which might be candidates if the i^{th} argument is of type t, i.e., the set

$$\{(t_1, \ldots, t_i, \ldots, t_c) \in F \mid t \preceq t_i\}. \tag{4.16}$$

At the resolution phase, the intersection of all c sets defined by (4.16) is computed by *AND*ing the bit-vector representation of these sets. The smallest multi-type in the intersection is then found using a *find-first-set* operation, which can often be implemented as a single machine instruction.

Assuming that the multi-methods in (4.12) are positioned in the following order:

$$\{(\mathsf{E}, \mathsf{D}), (\mathsf{B}, \mathsf{D}), (\mathsf{A}, \mathsf{D}), (\mathsf{A}, \mathsf{A})\},$$

then the bit-vectors assigned with the sets $T_1(F)$ and $T_2(F)$ of (4.13) are shown in Table 4.5.

$T_1(F)$	vector
A	0011
B	0100
E	1111

$T_2(F)$	vector
A	0001
D	1111

Table 4.5: SRP representation for the multi-family of (4.12)

4.7.3 Reducing the Space Requirement of the Mono-dispatch Stage with Type Slicing

We applied the mono-dispatch reduction on multiple dispatching benchmarks, drawn from various languages. The resulting hierarchies were used as benchmarks to single-dispatching algorithms. Degenerate multi-families, and degenerate arguments were removed.[3]

The mono-dispatch stage in SRP or CNT [5, 52, 77, 87] is currently carried out using either the technique of SC or RD for single dispatching, which are both null elimination schemes.

[3]A multi-family F is degenerate if $|F| = 1$. The i^{th} argument is degenerate if $|T_i(F)| = 1$.

Table 4.6 compares the average number of *bits* per multi-family for the mono-dispatch stage and the resolution stage. The *mono-dispatch stage* is carried out using either our type slicing (TS) technique, or using an ideal null elimination scheme which requires w entries. The *resolution stage* is carried out using either SRP or CNT. The results were broken down by arity of the multi-method, which ranged between 2 to 4.

Hierarchy	Arity											
	2				3				4			
	TS	w[a]	SRP	CNT	TS	w[a]	SRP	CNT	TS	w[a]	SRP	CNT
Cecil	296	718	234	110	380	1,718	168	501	263	2,798	16	16
Dylan	228	1,100	115	142	496	3,903	609	8,906	697	4,031	801	38,475
Cecil-	269	177	180	473	327	137	30	73	408	272	16	16
Cecil2	241	373	270	644	286	740	30	73	352	272	16	16
Harlequin	283	466	148	185	284	471	123	238	0	0	0	0
Vor3	330	303	347	925	485	278	666	1,449	328	320	16	16
Vortex3	351	2,100	294	720	469	5,996	302	828	472	320	16	16

[a]An ideal null elimination scheme

Table 4.6: Average number of bits per family for the mono-dispatch stage and the resolution stage

The space requirements presented in the table are in a way a lower bound, since we used a bit granularity rather than byte. For instance, in the ideal null elimination scheme, an entry for family F occupies $\lceil \log_2 |F| \rceil$ bits. Also, in the c-dimensional matrix of CNT, the number of bits in one matrix entry is not necessarily divisible by 8. The same is true for the bit-vector size in SRP, or the size of entries in our array implementation of TS. Therefore, shift and mask operations are needed in order for the assumption to hold.

We observe the following in the table:

1. The *relative advantage* of SRP over CNT (in the *resolution stage*) increases with the arity. For example, in the DYLAN hierarchy SRP improves on CNT by 19% for an arity of 2, by 93% for an arity of 3, and by 98% for an arity of 4. This fact is in agreement with the theoretical analysis of SRP in (4.15) and of CNT in (4.14).

2. The space requirement of the *mono-dispatch stage* using an ideal null elimination scheme dominates those of the *resolution stage* using SRP. In other words, the benefits of a space efficient resolution stage are wasted if we simply use RD or SC in the mono-dispatch stage.

   The reason that null elimination performs so poorly in the multiple dispatching benchmark is that many multi-methods have *root-type arguments* to handle unexpected combination of arguments. Null elimination schemes cannot compress such multi-methods. Therefore, it was even suggested [77] to compare different algorithms on data sets without such multi-methods. Duplicates elimination schemes, such as CT and our TS, performs especially well on such cases.

3. Using TS instead of a null elimination scheme reduces, in most cases, the space requirement of the *mono-dispatch stage*. In the Cecil- and Vor3 hierarchies (and

in the Cecil2 hierarchy for the case of an arity 4) an ideal null elimination scheme is better than TS. However, in the other five hierarchies TS is better by as much as 92%.

4.8 Incremental Algorithm for Constant-time Subtyping Tests

4.8.1 Problem Definition

In the *incremental version* of the subtyping problem, the type hierarchy may grow during program execution as new types are added as leaves. Such additions are allowed, e.g., in JAVA [7]. This dynamic hierarchy model gains increasing popularity since it shortens the initialization time of applications loaded from a local storage device, such as a disk, and even more so from a remote device such as the network. Also, in mission critical systems, in which an application cannot be restarted, it is convenient to make updates to the running software by simply loading more types. Finally, we note that this problem also appears in the implementation of ScopedMemory of Real-time Java [34, 110] where the memory is organized at runtime in a dynamic tree structure.

Almost all previous work on the subtyping problem [59, 84, 90, 115, 127, 133] mention an incremental extension of the proposed algorithm. However, these after thought additions invariably suffer from the limitation that the total time for building the associated data structures is much greater in a piecemeal feed of the type hierarchy, than if the entire hierarchy is supplied up front.

An algorithm for the dispatching problem is also a solution to the subtyping problem, since if we associate with each type t a unique family $F_t = \{t\}$, then $a \preceq b$ holds precisely when $\mathsf{dispatch}(F_b, a) \neq \mathrm{null}$. We know of no opposite reduction. Indeed, solutions of the subtyping problem tend to be more efficient then their dispatching counterparts.

After this reduction, applying TS gives constant-time subtyping tests. The reason is that the dispatch time is $O(\log|F|)$, and $|F| = 1$. For completeness we describe the subtyping algorithm in detail.

4.8.2 Previous Work on Subtyping Tests

(B)PE: *(Bit) Packed Encoding* [133] SC was specialized into a subtyping test scheme called Packed Encoding (PE), by Vitek, Horspool and Krall. They also suggested packing several identifiers into the same byte, resulting in an encoding called Bit Packed Encoding (BPE).

NHE: *Near Optimal Hierarchical Encoding* [90] *Bit-vector encoding* embeds the hierarchy in the lattice of subsets of $\{1, \ldots, \beta\}$. In this scheme, each type a is encoded as a vector vec_a of β bits, such that relation $a \preceq b$ holds iff

$$\mathrm{vec}_b \wedge \mathrm{vec}_a = \mathrm{vec}_b\,. \tag{4.17}$$

The challenge in building a bit-vector encoding is in finding the minimal β for which such an embedding is possible. The problem is NP-hard [75], but several good heuristics were proposed. Currently, NHE due to Krall, Vitek and Hoorspool, is the best (in terms of smallest β) algorithm for bit vector encoding.

Bommel and Beck [127] describe an incremental technique for updating a bit-vector encoding. Although no asymptotic results are given, and testing was limited to "randomly generated hierarchies", it appears from the authors description that the technique is useful for small hierarchies, with at most 300 types.

PQE: *PQ-Encoding* [136] PQ-encoding, which uses PQ-trees [17] gives one of the best compression results of the subtyping matrix, while maintaining constant time for queries. PQE is not incremental since it requires feeding whole program information into a very sophisticated data structure.

Dynamic subtyping in single inheritance Dietz [40, 41] suggested an asymptotically optimal solution to the dynamic subtyping problem, i.e., linear space requirement and constant time for queries and additions. The idea is to maintain the pre- and post-orders of the tree in an *ordered list* (see Section 4.10). Subtyping tests are answered by using two ORDER queries relying on the fact that $a \preceq b$ iff a occurs before b in the post-order and b occurs before a in the pre-order.

A different incremental algorithm for single inheritance is Cohen's algorithm [29]. Let $l_t = |\text{ancestors}(t)|$ denote the level of t. The algorithm associates with each type t an array of length l_t, storing the type-id of each $t' \succeq t$ in position $l_{t'}$. Cohen's algorithm gives *simple* and constant-time subtyping tests. The cost is that the space requirement might be $O(n^2)$ if the hierarchy is, for instance, a long chain. In practice, since the maximal number of ancestors is relatively small, the space requirement of Cohen's encoding is tolerable. Jalapeño [4], IBM implementation of the JAVA virtual machine (JVM), uses Cohen's algorithm for subtyping tests where the supertype is a class.

4.8.3 Subtyping using Type Slicing Scheme

Our incremental subtyping algorithm is based on the order-preserving heuristic for maintaining the slices (described in Section 4.10). The non-incremental variant is described next.

Figure 4.5 showed the slicing of the running example into two slices. We associate with type A, for example, the following data,

$$\begin{aligned} s_{\mathsf{A}} &= 1, \\ \text{id}_{\mathsf{A}} &= 2, \\ D_1(\mathsf{A}) &= [2, 4], \\ D_2(\mathsf{A}) &= [1, 3]. \end{aligned} \tag{4.18}$$

Encoding a hierarchy in this fashion requires at most $2\kappa n + 2n$ memory cells.

Since $\mathrm{descendants}(t) = \bigcup_{1 \le i \le \kappa} D_i(t)$, we have that $a \preceq b$ holds if and only if the position of a is within the appropriate interval of b, i.e.,

$$\mathrm{id}_a \in D_{s_a}(b). \tag{4.19}$$

For instance, we test whether $\mathsf{G} \preceq \mathsf{A}$, by retrieving the slice of G, $s_{\mathsf{G}} = 1$, and its identifier, $\mathrm{id}_{\mathsf{G}} = 4$. We then determine whether this identifier falls inside the appropriate interval of A. In this example, we conclude that $\mathsf{G} \preceq \mathsf{A}$ since $4 \in [2, 4]$.

4.8.4 Experimental Results for Type Slicing

To make the comparison of incremental and non-incremental algorithms meaningful, we do not include in the space requirement pointers or other auxiliary data used in computing the encoding or in maintenance of the dynamic data structure. In the case of our *type slicing* (TS) algorithm this auxiliary data is a small number of (about four) words per type.

The *BTS variant* of the basic TS algorithm applies *bit packing* to compress the identifiers of types in small slices, in a manner similar to BPE. Note that the BTS and the BPE variants are slower than their non-packing counterparts, since they are obliged to use shifts and masks to unpack the type identifiers. As mentioned in Section 4.6, the superscript PQ shall denote the variant which use the PQ heuristic.

Creation time Table 4.7 compares both the total and the per-type run time of different subtyping algorithms on modern computing platforms. In the worst case hierarchy (Geode), the average time required to insert a type using (B)TS algorithms is as little as 16 micro-seconds. We also see that the PQ variants of TS are very slow, requiring 17.223 mSec *per type* in this hierarchy, whereas the basic TS algorithms require just a little more (21 mSec) to process the *entire* hierarchy.

To estimate the cost of using the PQE in an incremental fashion, we can compare the *total time* of PQE with the *per-type time* of the incremental (B)TS. In doing so we find that (B)TS is three to four orders of magnitude faster than PQE. Even the *total runtime* of the (B)TS algorithms is, on average, three times faster than that of PQE.

Despite the fact that the data on the NHE runs was generated on a different architecture, we argued [136] that PQE is in general faster than NHE.

Space requirement The main metric of subtyping algorithms is the encoding length, i.e., the number of bits per type. Table 4.8 compares the encoding length obtained by TS and its three variants with those of some other algorithms over the standard ensemble of 13 multiple inheritance hierarchies.

In comparing the last two columns of the table we learn that our quick order-preserving heuristic can be improved, sometimes by as much as 40% by applying the PQ heuristic. However, in going through the BTS column we discover that bit-packing is a more effective compression technique, outperformed by $\mathrm{TS}^{\mathrm{PQ}}$ in only two out of the 13 hierarchies. Therefore, it seems worthwhile to spend the little extra time in the subtyping tests of BTS.

Hierarchy	PQE [a]		NHE [b]		(B)PE [c]		**(B)TS** [a]		**(B)TS^{PQ}** [a]	
	Total (mS)	Per $\mathcal{T}$ (μS)	Total (mS)	Per $\mathcal{T}$ (μS)	Total (mS)	Per $\mathcal{T}$ (μS)	Total (mS)	Per $\mathcal{T}$ (μS)	Total (mS)	Per $\mathcal{T}$ (μS)
IDL	1	15	-	-	5	75	0.1	1	1	15
Laure	3	10	21	71	9	31	0.5	2	90	305
Unidraw	1	2	93	151	10	16	1.6	3	90	147
JDK 1.1	1	4	19	84	10	44	0.3	1	30	133
Self	48	27	1,367	759	22	12	20.2	11	12,100	6,715
Ed	29	67	136	313	12	28	1.7	4	711	1,638
LOV	42	96	168	385	10	23	2.0	5	941	2,158
Eiffel4	146	73	-	-	29	15	19.5	10	11,400	5,703
Geode	311	236	1,902	1,443	28	21	20.6	16	22,700	17,223
JDK 1.18	15	9	-	-	26	15	10.0	6	2,520	1,479
JDK 1.22	81	19	-	-	77	18	38.1	9	32,500	7,490
JDK 1.30	113	21	-	-	90	17	53.8	10	49,800	9,158
Cecil	24	26	-	-	13	14	4.4	5	2,000	2,146
Total	815	42	-	-	341	17	172.8	9	134,883	6,880
Median	29	21	136	313	13	18	4.4	5	2,000	2,146

[a]900 Mhz Pentium III
[b]500 Mhz 21164 Alpha
[c]750 Mhz Pentium III, user time in Linux

Table 4.7: Total time (in mSec) and average time per type (μSec) for generating a subtyping encoding

Hierarchy	PQE	NHE	BPE	PE	**BTS^{PQ}**	**BTS**	**TS^{PQ}**	**TS**
IDL	0	17	32	96	56	64	40	56
Laure	6	23	63	128	72	80	88	152
Unidraw	2	30	63	96	72	72	88	88
JDK 1.1	1	19	32	64	64	64	56	56
Self	39	53	126	344	88	96	120	152
Ed	36	54	94	216	144	152	376	408
LOV	42	57	94	216	144	152	376	408
Eiffel4	65	72	157	312	160	176	312	344
Geode	80	95	157	408	248	264	600	632
JDK 1.18	25	39	94	128	104	112	152	184
JDK 1.22	36	62	157	184	152	168	280	312
JDK 1.30	41	65	188	216	160	192	280	376
Cecil	22	58	94	192	104	112	184	216
Total	40	61	145	227	144	161	266	315
Median	36	54	94	192	104	112	184	216
Minimum	0	17	32	64	56	64	40	56
Maximum	80	95	188	408	248	264	600	632

Table 4.8: The encoding lengths of different subtyping algorithms

Note also that applying the PQ heuristic on top of bit packing does not yield much: the maximal compression of the encoding length in doing so is 16.7%. Therefore, our basis of comparison of the incremental algorithms with their static counterparts will be the BTS column.

The BTS encoding is better than PE in all hierarchies, but is only better than BPE in the Self hierarchy. It is slightly worse than BPE in all but the Geode hierarchy. BTS does not yield as good encoding length as NHE and PQE. However, since BTS is incremental, it can answer subtyping queries during any stage of the creation process—a task in which PE, BPE, NHE and PQE fail.

4.9 Open Problems

The most important problem this thesis leaves open is an incremental dispatching algorithm, i.e., allowing additions of types, along with their methods, at the bottom of the hierarchy. Another natural extension worth investigating is in allowing also *deletion* of leaves from the hierarchy, as supported, at least in part, by JAVA. Other extensions include addition of new methods to existing types, or as it might be the case in knowledge representation, reasoning, database management, and query processing, allowing insertion of types anywhere in the hierarchy.

In the more pure algorithmic front, it would be both interesting and useful to generalize the PQ-tree data structure to support modifications of existing constraints when a new element is added to the universe.

Our algorithms assumed that ambiguities are resolved by an appropriate augmentation of families. Some OO languages resolve ambiguities based on a *linearization* of the partial order $\preceq$. COMMONLOOPS [16], for example, uses a global type ordering, while CLOS [15] uses a local type ordering. Extending our algorithms to support linearization based ambiguity resolution appears to be a worthy prospect.

Dispatching and linearization also occur in JAVA exception handling, as the following code excerpt shows.

```
try {...}
catch(D d) {...}
catch(E e) {...}
catch(A a) {...}
```

When an object o of a dynamic type $a = a(o)$ is thrown in a **`try`** block, the program executes the first **`catch`** block whose argument is a supertype of a. Thus, each of the **`catch`** clauses is a subtyping test. When the number of such clauses is large, it might be worthwhile to choose the exception handler using a dispatching algorithm which will find the clause with the smallest supertype.[4] Ambiguities are resolved using the order of the **`catch`** blocks chosen by the programmer.[5]

[4]The above code, if read in C++ [124], leads to the same problem. This is due to the separate compilation model of C++, in spite of the fact that exceptions are caught according to the *static* type of the thrown object.

[5]In fact, there is no possibility for ambiguity in JAVA exception handling. The reason is that a type in

4.10 An Order-Preserving Heuristic for Finding the Slices

The algorithm for creating the slices uses the order-preserving heuristic as an internal procedure in the following fashion. We traverse the types in a topological order, i.e., as if the hierarchy is given to us incrementally where new types can be added only as leaves. For each such type we try to find the first slice it can be added to, without violating the slicing property. If no such slice is found, we create a new slice.

Given a slice $\mathcal{T}_i$ and a type t, we give an algorithm whose runtime is $|\mathrm{ancestors}(t)|$, which checks whether there is a valid list location for inserting t, and if so, finds it. The idea is to maintain an *ordered list* for all types in a slice. The slicing property is slightly modified so that the sets $D_i(t)$ are consecutive in *the ordered list* of slice $\mathcal{T}_i$.

An *ordered list* is a data structure supporting two kinds of operations: INSERT transactions and ORDER queries of the following sort. Given two positions in the list (usually as pointers to list nodes), determine which one precedes the other. In a paper entitled "Two Algorithms for Maintaining Order in a List", Dietz and Sleator [41] give the best algorithm for this problem, achieving $O(1)$ worst-case time per operation. However, the authors comment that their other algorithm "is probably the best algorithm to use in practice", even though it is theoretically inferior, since its amortized[6] insertion time is $O(\log n)$. This other algorithm is based on a technique known as *self-adjustment*. In a nutshell, each list node is assigned an integer position in an increasing order, thus ORDER queries are answered in constant time. "Holes" are left to support future insertions, and if a "hole" is filled, then we redistribute the positions in some "sufficiently large and uneven" list interval. We implemented this simple algorithm and indeed found it to be very fast in practice.

Before describing the order-preserving heuristic we need to make the notions of list locations and list intervals more precise.

Definition 4.4 *A* location *of a linked list is either (i) the beginning of the list, (ii) the end of the list, or (iii) any point between two consecutive nodes of the list. An* interval *in the list is a set of consecutive locations. The* boundary *of an interval comprises its first and last locations. All other locations are called the* interior *of the interval.*

The boundary usually contains two locations, the first and the last. For example, the interval marked as $D_1(\mathsf{A})$ in Figure 4.7 has two interior locations and two boundary locations.

The interior of *degenerate intervals* is empty; in such intervals the first and last locations are the same. An *empty interval* has an empty boundary and an empty interior.

Definition 4.5 *The* interval of the set $D_i(t)$ *in the ordered list of* $\mathcal{T}_i$ *includes* all *locations in the sub-list defined by* $D_i(t)$.

the **`catch`** block must be a subtype of the **`class`** `Throwable`, and JAVA has a single inheritance **`class`** hierarchy (and ambiguities cannot occur in a single inheritance hierarchy).

[6]The *amortized time* of an operation is $c(n)$, if a sequence of n operations requires at most $nc(n)$ time. The worst case time of any single operation can however be much greater than $c(n)$.

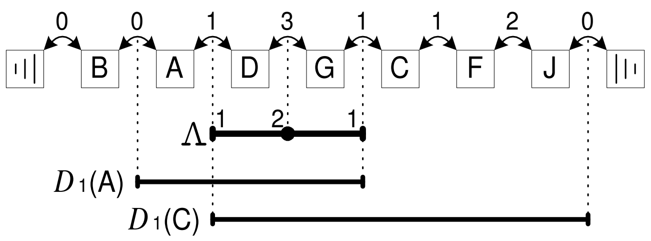


Figure 4.7: Addition of a new type to the first slice of Figure 4.5

In other words, the interval of the set $D_i(t)$ also includes the location prior to the first element of $D_i(t)$, as well as the location following its last element.

In the example, we see in Figure 4.5 that A has three descendants in the first slice, i.e., $D_1(\mathsf{A}) = \{\mathsf{A}, \mathsf{D}, \mathsf{G}\}$. In Figure 4.7 we see that these three types are consecutive in the ordered list of the first slice and that the interval of $D_1(\mathsf{A})$ has four locations.

When inserting a new type t to the ordered list of $\mathcal{T}_i$, we search for a list location where inserting t will not violate the slicing property. Such locations must belong to the interval of $D_i(t')$ for *all* ancestors t' of t, i.e., $t' \succeq t$. Let $\mathcal{I}$ denote the set of all such intervals, and let Λ denote the intersection of all intervals in $\mathcal{I}$. A list locations in Λ is called a *candidate* for inserting t.

Algorithmically, Λ is computed by finding the largest first location of the intervals in $\mathcal{I}$, and the smallest last location of these intervals. (Comparisons are carried out using simple ORDER queries.) If Λ is empty, then we conclude that t cannot be inserted into $\mathcal{T}_i$. The time for computing the intersection and for checking whether it is empty is in the following asymptotic growth class:

$$O(|\mathrm{parents}(t)|) \subseteq O(|\mathrm{ancestors}(t)|).$$

It is also required that t does not "break" any interval of $D_i(t")$, $t" \not\succeq t$. More precisely, a location is an *invalid candidate* if it belongs to the *interior* of these intervals. Although it is possible to check each candidate location $\ell \in \Lambda$ against every interval of a type $t" \in \mathcal{T} \setminus \mathrm{ancestors}(t)$, the running time of this exhaustive search may be linear in the size of the hierarchy!

Figure 4.7 shows the ordered list of the first slice of Figure 4.5. We try to insert to that slice a new type whose parents are A and C. We see the intervals of $D_1(\mathsf{A})$ and $D_1(\mathsf{C})$, and their intersection Λ. The new type can only be inserted in a candidate location $\ell \in \Lambda$. The candidate location between types D and G, for example, is invalid since it belongs to the interior of the interval of $D_1(\mathsf{D})$, and D is not an ancestor of the new type. The other two candidate locations are valid.

The *counts* λ_ℓ associated with each location in Figure 4.7 are a part of a more efficient implementation for determining if a location is an invalid candidate. For each location ℓ in the ordered list, let λ_ℓ be the number of all intervals $D_i(t)$, such that ℓ is in the *interior* of $D_i(t)$. For instance, the location between types D and G has a count of 3, since it is in the interior of $D_1(\mathsf{A})$, $D_1(\mathsf{C})$ and $D_1(\mathsf{D})$.

A location ℓ in the *interior* of Λ is contained in the interior of *all* intervals defined by $D_i(t')$, $t' \succeq t$. Therefore, for all candidate locations $\ell \in \Lambda$ we have that

$$\lambda_\ell \geq |\text{ancestors}(t)|. \tag{4.20}$$

The location is an invalid candidate if it is contained in the interior of any other interval, and therefore

$$\lambda_\ell > |\text{ancestors}(t)|. \tag{4.21}$$

In the example of Figure 4.7, the location between types D and G is an invalid candidate, since its count is strictly higher than the number of ancestors.

We must be more careful in checking a location ℓ in the *boundary* of Λ. Let $I \in \mathcal{I}$ be arbitrary. Then, by definition $\ell \in I$. It is not however guaranteed that ℓ is in the interior of I. We therefore compute the number n_ℓ of intervals $I \in \mathcal{I}$ such that ℓ is in the interior of I. A boundary location ℓ is an invalid candidate iff

$$\lambda_\ell > n_\ell. \tag{4.22}$$

In our example, both boundary locations are valid candidates.

Although there are several special cases and many nitty-gritty details, it is a straightforward matter to update in $O(1)$ time the counts λ_ℓ with every insertion. (Note that the count may change only for two locations: before and after the insertion point.) Also, computing n_ℓ and checking (4.22) can be done in $O(|\text{ancestors}(t)|)$ time. It is potentially more time consuming to do the check (4.21) since we have no a priori bounds on $|\Lambda|$.

Non-exhaustive techniques for finding a valid insertion location We found empirically that if t could not be inserted at the boundary of Λ, then it was rarely possible to insert it to the interior of Λ. For example, out of the 4339 types of JDK 1.22, only 22 types (less than 0.5%) were inserted in the interior of Λ. In all other hierarchies of our data set, the total number of such types was even smaller, and their fraction was always lower than 1%.

Therefore, it does not seem necessary to apply the check (4.21) at all. Nevertheless, we should note that there are ways of implementing (4.21) more efficiently than an exhaustive search. It follows from (4.20) and (4.21) that there exists a valid location ℓ in the interior of Λ if and only if

$$\min\{\lambda_\ell \mid \ell \text{ is in the interior of } \Lambda\} = |\text{ancestors}(t)|.$$

Therefore, the problem of finding a valid location in the interior of Λ is reduced to the famous *range minima* problem [65]. A simple solution to the range minima problem is to maintain a balanced binary search tree (BBST) over the ordered list of $\mathcal{T}_i$, such that each internal node in it stores the minimum of λ_ℓ of all locations ℓ in the subtree rooted at this node. This representation adds $O(\log n)$ time to each insertion operation. It is standard to use this BBST to compute the minimum of any given interval. More sophisticated solutions to the range minima problem require only constant time per operation [65]. It is not clear whether these algorithms have any practical utility.

Inserting types with a single parent into multiple inheritance hierarchies Finally, we present an optimization for quickly inserting a type t with a *single* parent p. Let i be the slice of p, i.e., $p \in \mathcal{T}_i$. Consider the ordered list of $\mathcal{T}_i$, and a list location ℓ immediately to the left (or to the right) of p. We claim that ℓ is valid for t. Assume the contrary, i.e., ℓ is in the *interior* of some interval defined by $D_i(t")$, and that $t"$ is *not* a supertype of t. Combined with the fact that p is adjacent to ℓ, we conclude that $p \in D_i(t")$, and therefore, $p \preceq t"$. Since $t \preceq p$, it follows that $t \preceq t"$, which contradicts our assumption.

Incremental subtyping algorithm Recall that each slice is kept in an *ordered list*. Instead of associating integer values with id_t and $D_i(t)$ as in (4.18), we now use pointers to cells in the ordered list. The test (4.19) can be carried out in constant time using two ORDER queries. We show next how to update this representation as new types are added.

When a type t is added to the ordered list of slice $\mathcal{T}_i$, only the list intervals of its ancestors can change. Therefore, for each $t' \succeq t$ we check if t was added at the boundary of $D_i(t')$, and if so update it. Updating all list intervals $D_i(t')$ takes $O(|\mathrm{ancestors}(t)|)$ time. Since the insertion time of the heuristic is $O(\kappa|\mathrm{ancestors}(t)|)$, the asymptotic time bound remains the same.

When a new slice is created, the arrays which store $D_i(t)$, $i = 1, \ldots, \kappa$, must be extended. Note that with the cost of a constant factor increase of the space requirement, the amortized time for extending an array is constant. Using techniques of "background copying" [42], the *worst case* time for an array extension operation becomes constant as well.

Chapter 5

Incremental Algorithms for Dispatching in Dynamically Typed Languages

Chapter Summary
A fundamental problem in the implementation of object-oriented languages is that of a frugal dispatching data structure*, i.e., support for quick response to dispatching queries combined with compact representation of the type hierarchy and the method families. Previous theoretical algorithms tend to be impractical due to their complexity and large hidden constant. In contrast, successful practical heuristics, including Vitek and Horspool's* compact dispatch tables *(CT) [132] designed for dynamically typed languages, lack theoretical support. In subjecting CT to theoretical analysis, we are not only able to improve and generalize it, but also provide the first non-trivial bounds on the performance of such a heuristic.*

Let n, m, ℓ denote the total number of types, messages, and different method implementations, respectively. Then, the dispatching matrix, whose size is nm, can be compressed by a factor of at most $\iota \equiv (nm)/\ell$. Our main variant to CT achieves a compression factor of $\frac{1}{2}\sqrt{\iota}$. More generally, we describe a sequence of algorithms CT_1, CT_2, CT_3, ..., where CT_d achieves compression by a factor of (at least) $\frac{1}{d}\iota^{1-1/d}$, while using d memory dereferencing operations during dispatch. This tradeoff represents the first bounds on the compression ratio of constant-time dispatching algorithms.

A generalization of these algorithms to a multiple inheritance *setting, increases the space by a factor of $(2\kappa)^{1-1/d}$, where κ is a metric of the complexity of the topology of the inheritance hierarchy, which (as indicated by our measurements) is typically small.*

The most important generalization is an incremental *variant of the CT_d scheme for a single inheritance setting. This variant uses at most twice the space of CT_d, and its time of inserting a new type into the hierarchy is optimal. We therefore obtain algorithms for efficient management of dispatching in dynamic-typing, dynamic-loading languages, such as* SMALLTALK *and even the* JAVA `invokeinterface` *instruction.*

Message dispatching stands at the heart of object-oriented programs, being the only way objects communicate with each other. To implement dynamic binding during dispatch, the runtime system of object-oriented languages uses a *dispatching data structure*, in which a *dispatching query* finds the appropriate implementation of the message to be called, according to the dynamic type of the message receiver. A fundamental problem in the implementation of such languages is then a frugal implementation of this data structure, i.e., simultaneously satisfying (i) compact representation of the type hierarchy and the families of different implementations of each method selector, and (ii) quick response to dispatching queries.

Virtual function tables (VFT) are a simple and well known (see e.g., [123]) incremental technique which achieves dispatching in constant time (two dereferencing operations), and very good compaction rates. The VFT of each type is an array of method addresses. A location in this array represents a message, while its content is the address of an implementing method. The VFT of a subtype is an extension of the VFT of its supertype, and messages are allocated locations at compile time in sequential order. The static type of the receiver uniquely determines the location associated with each message. VFTs rely on *single inheritance*. *Multiple inheritance* implementations exist [68], but they are not as elegant or efficient.

The challenge in the dispatching problem is therefore mostly in dealing with *dynamically typed* and/or multiple inheritance languages. Also very important is the *incremental* version of this problem, in which types (together with their accompanying messages and methods) are added at the bottom of the hierarchy.

Our contribution (described in greater detail in Section 5.3) includes a provable tradeoff between space and dispatching time with extensions to multiple inheritance hierarchies. The pinnacle of the results is an incremental algorithm for maintaining a compact dispatch table in dynamically typed languages.

Outline The remainder of this chapter is organized as follows. The subtyping tests problem is defined in Section 5.1. Some straightforward solutions for this problem are described in Section 5.2. Section 5.3 mentions our results in perspective of these solutions. Section 5.4 presents the generalized CT schemes for single inheritance hierarchies. Section 5.5 shows how these schemes can be made incremental. A (non-incremental) version of these schemes for multiple inheritance hierarchies is described in Section 5.6. Section 5.7 presents the experimental results: timing and compression values on a dataset of 35 hierarchies collected from both single and multiple dispatching languages. Open problems and directions for future research are the subject of Section 5.8.

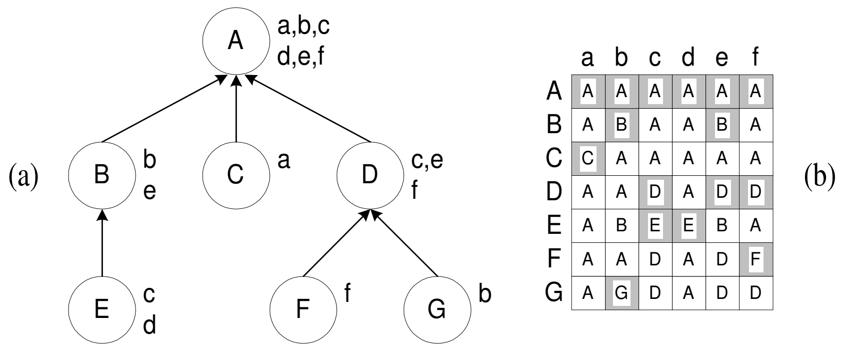
5.1 Problem Definition

We define the dispatching problem in a similar fashion to the *colored-ancestors* abstraction described by Ferragina and Muthukrishnan [60]: a *hierarchy* is a partially ordered set $(\mathcal{T},\preceq)$ where $\mathcal{T}$ is a set of types and $\preceq$ is a reflexive, transitive and anti-symmetric *subtype relation*. The $\min$ operator return the set of smallest types in any given set:

$$\min(X) = \{t \in X \mid \nexists t' \in X : t' \neq t, t' \preceq t\}.$$

Let $F \subseteq \mathcal{T}$ denote the *family* of types which have a *method implementation* for the same *message*.[1]

For example, consider the single inheritance hierarchy in Figure 5.1a. Type names are uppercase and messages are lowercase, e.g., type D implements the messages c, e and f. Then, $\{\mathsf{A}, \mathsf{D}, \mathsf{E}\}$ is the family of method implementations of c.



	a	b	c	d	e	f
A	A	A	A	A	A	A
B	A	B	A	A	B	A
C	C	A	A	A	A	A
D	A	A	D	A	D	D
E	A	B	E	E	B	A
F	A	A	D	A	D	F
G	A	G	D	A	D	D

Figure 5.1: (a) A small example of a single inheritance hierarchy, and (b) its dispatching matrix

Given a family F and a type t, $\mathsf{cand}(F, t)$ is the set of candidates in F, i.e., those ancestors of t in which an implementation of the given message exists:

$$\mathsf{cand}(F, t) \equiv F \cap \text{ancestors}(t). \tag{5.1}$$

In the figure, we have for example $\mathsf{cand}(\{\mathsf{A}, \mathsf{D}, \mathsf{E}\}, \mathsf{G}) = \{\mathsf{A}, \mathsf{D}\}$.

A *dispatching query* $\mathsf{dispatch}(F, t)$ returns either *the smallest candidate* or null if no such unique candidate exists. (A null result represents either the *message not understood* or *message ambiguous* error conditions.) Specifically,

$$\mathsf{dispatch}(F, t) \equiv \begin{cases} t' & \text{if } \min(\mathsf{cand}(F, t)) = \{t'\}, \\ \text{null} & \text{otherwise.} \end{cases}$$

Definition 5.1 *Given a hierarchy* $(\mathcal{T}, \preceq)$ *and a family collection* $\mathcal{F} \subseteq \wp(\mathcal{T})$*, the* dispatching problem *is to* encode *the hierarchy in a data structure supporting* $\mathsf{dispatch}(F, t)$ *queries for all* $F \in \mathcal{F}$*,* $t \in \mathcal{T}$*.*

A solution to the dispatching problem is measured by the following three metrics: (i) space, (ii) query time, and (iii) encoding creation time. We would like to express these as a function of the following problem parameters $\langle n, m, \ell \rangle$: the number of types, families, and implementations (or family members). Specifically,

$$\begin{aligned} n &\equiv |\mathcal{T}|, \\ m &\equiv |\mathcal{F}|, \\ \ell &\equiv \sum_{F \in \mathcal{F}} |F|. \end{aligned} \tag{5.2}$$

[1]We abstract away from the nomenclature of different languages, and use the terms *message* (also called selectors or signature) for the unique identifier of a family of *implementation* (also called methods, member functions, operations, features, etc.)

In Figure 5.1 for example, we have $n = 7$, $m = 6$ and $\ell = 16$.

The incremental version of the problem, is to maintain this data structure in the face of additions of types (with their accompanying methods) to the bottom of the hierarchy, as done in languages such as JAVA [7].

5.2 Straightforward Solutions

The most obvious solution is an $n \times m$ *dispatching matrix*, storing the outcomes of all possible dispatching queries. Figure 5.1b shows the dispatching matrix of Figure 5.1a, where the ℓ grey entries correspond to (non-inherited) family members.

In the dispatching matrix representation, queries are answered by a quick indexing operation. However, the space consumption is prohibitively large, e.g., 512MB for the dispatching matrix in the largest hierarchy in our benchmarks (8,793 types and 14,575 families).

Note that an encoding that does not try to compress pointers must use at least ℓ cells for representing the ℓ different method addresses. We would like to get as close as possible to this space requirement while preserving a constant and small query time. The dispatching matrix can be potentially compressed by a factor of

$$\iota \equiv (nm)/\ell. \tag{5.3}$$

We shall refer to ι as the *optimal compression factor*, and to schemes attempting to reach ι as *duplicates-elimination schemes*. In our data-set of 35 large hierarchies (see Section 5.7), $\iota \approx 725$.

Let w denote the number of non-null entries in the dispatching matrix, i.e.,

$$w \equiv |\{\langle F, t\rangle \mid \mathsf{dispatch}(F, t) \neq \mathrm{null}\}| . \tag{5.4}$$

By eliminating null memory cells, the dispatching matrix might be compressed by a factor of $(nm)/w$, which is around 150 in our data-set. Examples of null-elimination schemes are *row displacement* [45, 47], *selector coloring* [44, 118], and *virtual function tables* (VFT) [123]. In single inheritance and static-typing setting of the problem, the VFT technique uses precisely w memory cells.

In the more general setting, the matrix can also be compressed into $O(w)$ cells (with fairly large constants) by using perfect hashing [64] or one of its variants. Even though dispatching time is constant in perfect hashing, it is complicated by the finite-field arithmetic incurred during the computation of the hash function.

With additional increase to the complexity of dispatching, there are variations to the famous FKS [64] scheme which use $w + o(w)$ cells. There is also a dynamic version of perfect hashing [42] which can support incremental dispatching. The memory toll is even larger, with constants in the range of a thousand.

Notice that even complete null-elimination gives suboptimal compression, since w might be substantially larger than ℓ. In our benchmark of 35 large hierarchies, w/ℓ is on average 8.3, and in one hierarchy it is 122.4!

It is not difficult to come close to complete duplicates-elimination, i.e., a space of $O(\ell)$, with a simple representation of the hierarchy as a graph where types are nodes and immediate inheritance relations are edges. The cost is of course the query time, which becomes $O(n)$, since we must traverse all the ancestors of a receiver in order to find the nearest family member. Sophisticated caching algorithms (as employed in the runtime system of SMALLTALK [37]) make the typical case more tolerable than what the worst case indicates.

5.3 Our Result in Perspective

There is a large body of research on the dispatching problem (see e.g., [37, 44, 45, 47, 78, 118, 131, 132, 135, 137]). The focus in these was on "practical" algorithms, which were evaluated empirically, rather than by provable upper bound on memory usage. The main theoretical research on the topic [60, 100] produced algorithms (for the single inheritance setting) which using minimal space ($O(\ell)$ cells) supported dispatching in doubly logarithmic, $O(\lg\lg n)$, time. However, the hidden constants are large, and the implementation is complicated.

In this chapter, we describe a different tradeoff: constant-time dispatching in d steps, while using at most $d\ell\sqrt[d]{\iota}$ cells. Stated differently, our results are that d steps in dispatching (provably) achieve a compression rate of $\frac{\iota}{d\sqrt[d]{\iota}}$. For example, with $d = 2$ the compression is by a factor of at least half of the square root of ι, the optimal compression rate. Also, the compression factor is close to optimal, $\frac{\iota}{2\lg m}$, when the dispatching time is logarithmic, $\lg m$.

An important advantage of these results in comparison to previous theoretical algorithms is that they are simple and straightforward to implement, and bear no hidden constants. In fact, our algorithms are based on a successful practical technique, namely *compact dispatch tables* (CT), which was invented by Vitek and Horspool [132]. Viewed differently, the results presented here give the first proof of a non-trivial upper bound on practical algorithms.

Even though the algorithms carry on to multiple inheritance with the same time bounds of dispatching, the memory consumption increases by a factor of at most $(2\kappa)^{1-1/d}$, where κ can be thought of as a metric of the complexity of the topology of the inheritance hierarchy. (In a benchmark of 19 multiple inheritance hierarchies with 34,810 types, we found the median value of an upper bound for κ is 5, the average is 6.4, and the maximum is 18.) Our previous work [137] on dispatching gives an implementation of a dispatching data structure whose space was only $O(\kappa\ell)$, but the dispatching time was logarithmic. The results presented here complete the tradeoff spectrum, giving constant time dispatching with any number of steps. We give empirical evidence that the algorithms perform well in practice, in many cases even better than the theoretically obtained upper bounds.

We also describe an incremental version of the algorithms in a single inheritance setting, and prove that updates to the dispatching data structures can be made in optimal time. The cost is in a small constant factor increase (e.g., 2) to the memory footprint.

Readers may also take interest in some proof techniques, including the representation of dispatching as search in a collection of partitionings, the elegant Lemma 5.11, and the amortization analysis of the incremental algorithm.

5.4 Generalization of Compact Dispatch Tables for Single Inheritance Hierarchies

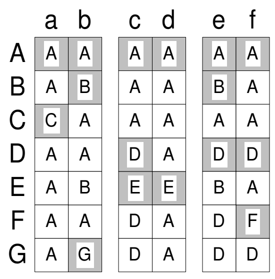
For simplicity, assume w.l.o.g. that the hierarchy is a tree (rather than a forest) rooted at a special node $\top \in \mathcal{T}$. There cannot be a *message ambiguous* in a single inheritance setting. To avoid the other error situation, namely *message not understood*, we assume that $\top \in F$ for all $F \in \mathcal{F}$. With this assumption, every dispatching query returns a single family member. The cost is in (at most) doubling the number of implementations ℓ. (At the end of this section we will show that the memory toll can be made much smaller.)

Vitek and Horspool's CT algorithm [132] partitions the family collection $\mathcal{F}$ into k disjoint *slices* $\mathcal{F} = \mathcal{F}_1 \cup \ldots \cup \mathcal{F}_k$. These slices break the dispatching matrix into k sub-matrices, also called *chunks*. The authors' experience was that chunks with 14 columns each give best results, and this number was hard-coded into their algorithm.

Figure 5.2 shows the three chunks of the dispatching matrix of Figure 5.1b for following partitioning:

$$\begin{aligned} \mathcal{F}_1 &= \{F_{\mathsf{a}}, F_{\mathsf{b}}\}, \\ \mathcal{F}_2 &= \{F_{\mathsf{c}}, F_{\mathsf{d}}\}, \\ \mathcal{F}_3 &= \{F_{\mathsf{e}}, F_{\mathsf{f}}\}. \end{aligned} \tag{5.5}$$

As Vitek and Horspool observed, and as can be seen in the figure, there are many identical rows in each chunk. Significant compression can be achieved by merging these rows together, and introducing, in each chunk, an auxiliary array of pointers to map each type to a row specimen.



	a	b	c	d	e	f
A	A	A	A	A	A	A
B	A	B	A	A	B	A
C	C	A	A	A	A	A
D	A	A	D	A	D	D
E	A	B	E	E	B	A
F	A	A	D	A	D	F
G	A	G	D	A	D	D

Figure 5.2: Three chunks of the dispatching matrix of Figure 5.1b

Why should there be many duplicate rows in each chunk? There are two contributing factors: (i) since the slices are small, there are not too many columns in a chunk, and (ii) that the number of distinct values which can occur in any given column is small, since, as empirical data shows, the number of different implementations of a selector is a small constant. Hence, there could not be too many distinct rows.

However, these considerations apply to any random distribution of values in the dispatching matrix. The crucial observation we make is that a much stronger bound on the number of distinct rows can be set relying on the fact that the values in the dispatching matrix are not arbitrary; they are generated from an underlying structured hierarchy.

Consider for example a chunk with two columns, with n_1 and n_2 distinct implementations in these columns. Simple counting considerations show that the number of distinct rows is at most $n_1 n_2$. Relying on the fact that the hierarchy is a tree we can show that the number of distinct rows is at most $n_1 + n_2$.

To demonstrate this observation, consider Figure 5.3a which focuses on the first chunk, corresponding to slice $\mathcal{F}_1 = \{F_{\mathsf{a}}, F_{\mathsf{b}}\}$.

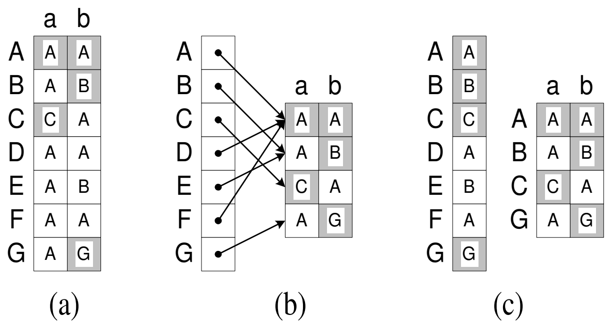


Figure 5.3: (a) The first chunk of Figure 5.1c, (b) the chunk compressed using an auxiliary array of pointers, and (c) the chunk compressed using an array of labels

As can be seen in the figure, the rows of types A, D, and F are identical. Figure 5.3b shows the compressed chunk and the auxiliary array. We see that this auxiliary array maps types A, D, and F to the same row.

We call attention to the (perhaps surprising) fact that it is possible to select from the elements of each row in Figure 5.3b a distinguishing representative. These representatives are members of what we call the *master-family*

$$F' = F_{\mathsf{a}} \cup F_{\mathsf{b}} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{G}\}.$$

The representatives of the four rows in the first chunk are A, B, C and G, in this order. The figure highlights these in grey. Also note that each member of the master-family serves as a representative of some row.

Figure 5.3c gives an alternative representation of the chunk, where each row is labeled by its representative. The auxiliary array now contains these labels instead of pointers. For example, the second row is labeled $\mathsf{B} \in F_{\mathsf{b}}$; the second and the fifth entry of the auxiliary array store B rather than the row specimen address.

Our improvement is based on the observation that the distinguishing representatives phenomenon is not a coincidence and on the observation that CT applies a *divide-and-conquer* approach to the dispatching problem: The search first determines the relevant master-family, and then continues to select the appropriate result among its members.

Let A_i denote the compressed i^{th} chunk of the dispatching matrix, and let B be the master dispatching matrix, whose columns are the auxiliary arrays of the chunks. Figure 5.4 shows matrices A_1, A_2, A_3 and B, which constitute the complete CT representation for the hierarchy of Figure 5.1. Note that the first column of B is the auxiliary array depicted in Figure 5.3c.

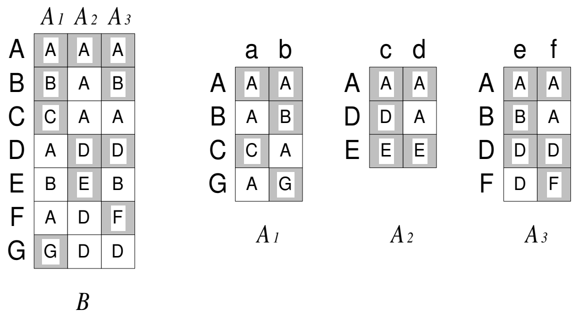


Figure 5.4: CT representation for the hierarchy of Figure 5.1

For each slice $\mathcal{F}_i$ let the *master-family* F'_i be the union of families in that slice, i.e., $F'_i \equiv \bigcup_{F \in \mathcal{F}_i} F$. Then, answering the query $\mathsf{dispatch}(F, t)$ at runtime requires three steps:

1. *Determine the slice of* F. That is, the family collection $\mathcal{F}_s$, such that $F \in \mathcal{F}_s$. If the partitioning into slices and the selector F are known at compile-time, as it is usually the case in dispatching of static-loading languages, then this stage incurs no penalty at runtime.

2. *Fetch the first dispatching result* $t' = \mathsf{dispatch}(F'_s, t)$. This value is found at the row which corresponds to type t and the column which corresponds to the master-family F'_s, i.e., $t' = B[t, s]$.

3. *Fetch the final dispatching result* $t'' = \mathsf{dispatch}(F, t)$. This type is found in the row of t' and the column of F in the compressed chunk A_s, i.e., $t" = A_s[t', F]$.

The algorithm merges together all the different messages in $\mathcal{F}_s$. At step 2, we find $t' \succeq t$, which is the smallest candidate in the merged master-family. Matrix B (of size $n \times k$) is the dispatching matrix of the types $\mathcal{T}$ and the master-family collection $\{F'_1, \ldots, F'_k\}$.

The search then continues with t', to find $t" \succeq t'$, the smallest candidate in F, the original family. Each matrix A_i (of size $|F'_i| \times |\mathcal{F}_i|$) is the dispatching matrix of the types in F'_i and the family collection $\mathcal{F}_i$.

To understand the space saving, consider just two families F_1 and F_2. The naive implementation of dispatch is using *two* arrays, each of size $n = |\mathcal{T}|$, which map each type t to two types $t"_1 \in F_1$ and $t_2" \in F_2$, such that $t_i" = \mathsf{dispatch}(t, F_i)$, $i = 1, 2$. A more compact representation can be obtained by using a *single* array of size n, to dispatch first on the merged master-family $F' = F_1 \cup F_2$. Let $t' \in F'$ be the result of this dispatch. The crucial point is that the smallest candidate for t', in either F_1 or F_2, is the same as for t. Since there are $|F'| \leq |F_1| + |F_2|$ different values of t', a continued search from t' (for either F_1 or F_2) can be implemented using two arrays, each of size $|F'|$. The first such array maps F' to F_1; the second to F_2. Total memory used is $n + 2|F'|$ instead of $2n$ cells, while the cost is an additional dereferencing operation.

More generally, given a dispatching problem for a family collection $\mathcal{F}$, the *CT reduction* partitions $\mathcal{F}$ into k disjoint slices

$$\mathcal{F} = \mathcal{F}_1 \cup \ldots \cup \mathcal{F}_k, \tag{5.6}$$

and merges together the families in each slice by defining a master-family

$$F'_i \equiv \bigcup_{F \in \mathcal{F}_i} F, \tag{5.7}$$

for all $i = 1, \ldots, k$. Let A_i be the matrix whose dimensions are

$$|F'_i| \times |\mathcal{F}_i|, \tag{5.8}$$

corresponding to the i^{th} slice. Then, the query $\mathsf{dispatch}(F, t)$ is realized by the fetch

$$A_s[\mathsf{dispatch}(F'_s, t), F], \tag{5.9}$$

where $F \in \mathcal{F}_s$.

Since both steps 2 and 3 in the dispatching are in essence a dispatching operation, better compaction of the dispatching data structure might be achieved by applying the CT technique recursively to either the matrix B, or all the matrices A_i. It is not difficult to see that each of the recursive applications will yield the same dispatching data structure, in which the set of selectors is organized in a three-level hierarchy of partitions: families, master-families, and master-master-families (so to speak). We chose to describe this 3-level system by applying the CT technique to the matrix B. The (potential) saving in space comes at a cost of another dereferencing step during dispatch. Clearly, we could recursively apply the reduction any number of times.

We need the following notation in order to optimize these recursive applications, i.e., find the optimal number of slices k, and the size of each slice. Let $\mathrm{mem}_d(n, m, \ell)$ denote the memory required for solving the dispatching problem of n types, m families and ℓ method implementations, using d dereferencing operations during dispatch. A simple dispatching matrix representation gives

$$\mathrm{mem}_1(n, m, \ell) = nm. \tag{5.10}$$

Each application of the CT reduction adds another dereferencing, while reducing a dispatching problem with parameters $\langle n, m, \ell \rangle$ to a new dispatching problem with parameters $\langle n, k, \ell' \rangle$, where

$$\ell' = \sum_{i=1}^{k} |F'_i| = \sum_{i=1}^{k} \left| \bigcup_{F \in \mathcal{F}_i} F \right|.$$

Note that $\ell' \leq \ell$. To see this recall that

$$\ell = \sum_{F \in \mathcal{F}} |F| = \sum_{i=1}^{k} \sum_{F \in \mathcal{F}_i} |F|,$$

and apply the fact that the cardinality of the union of sets is at most the sum of cardinalities of these sets

$$\ell' = \sum_{i=1}^{k} \left| \bigcup_{F \in \mathcal{F}_i} F \right| \leq \sum_{i=1}^{k} \sum_{F \in \mathcal{F}_i} |F| = \ell. \tag{5.11}$$

The reduction generates the matrices $A_1, \ldots, A_k$. To estimate their size suppose that all slices are equal in size, i.e., they all have x families. (For simplicity we ignore the case that m is not divisible by x, in which slices are *almost* equal.) Then, the total memory generated by the reduction is

$$\sum_{i=1}^{k} |F_i'| \times |\mathcal{F}_i| = \sum_{i=1}^{k} |F_i'| \times x = x \sum_{i=1}^{k} |F_i'| = x\ell' \leq x\ell.$$

To conclude, the costs of the CT reduction are another dereferencing and an additional space of $x\ell$. In return, a dispatching problem with parameters $\langle n, m, \ell \rangle$ is reduced to a new dispatching problem with parameters $\langle n, k, \ell' \rangle$, where $k = m/x$ and $\ell' \leq \ell$. Formally,

$$\mathrm{mem}_{d+1}(n, m, \ell) \leq \ell x + \mathrm{mem}_d(n, m/x, \ell), \tag{5.12}$$

where x is arbitrary.

Let CT$_d$ be the dispatching data structure and algorithm obtained by applying the CT reduction $d - 1$ times to the original dispatching problem. The recursion is ended by applying simple dispatching matrix at the last step. Thus, CT$_1$ is simply the dispatching matrix, while CT$_2$ is similar to Vitek and Horspool's algorithm (with $x = 14$). By making $d - 1$ substitutions of (5.12) into itself, and then using (5.10), we obtain

$$\mathrm{mem}_d(n, m, \ell) \leq \ell x_1 + \cdots + \ell x_{d-1} + \frac{nm}{x_1 x_2 \cdots x_{d-1}}, \tag{5.13}$$

where x_i is the slice size used during the i^{th} application of the CT reduction. Symmetry considerations indicate that the bound in (5.13) is minimized when all x_i are equal. We have,

$$\mathrm{mem}_d(n, m, \ell) \leq (d - 1)\ell x + \frac{nm}{x^{d-1}}, \tag{5.14}$$

which is minimized when $x = (nm/\ell)^{1/d}$.

Table 5.1 summarizes the space and time requirements of algorithms CT$_d$, where $\iota \equiv (nm)/\ell$ is the optimal compression factor.

The last row in the table is obtained by applying the CT reduction a maximal number of times. In each application the slice size is x (typically, $x = 2$). The collection $\mathcal{F}$ is then organized in a hierarchy of $\log_x m$ levels, which is also the number of dereferencing steps during dispatch. The memory used in each level is ℓx (see (5.12)).

The generalizations (Table 5.1) of CT$_d$ over Vitek and Horspool's algorithm is in the following directions: (i) a sequence of algorithms which offer a tradeoff between the

Scheme	Slice size	Time	Space	Compression factor
CT$_1$	N/A	1	$\ell\iota$	1
CT$_2$	$\sqrt[2]{\iota}$	2	$2\ell\sqrt[2]{\iota}$	$\frac{\iota}{2\sqrt[2]{\iota}}$
CT$_3$	$\sqrt[3]{\iota}$	3	$3\ell\sqrt[3]{\iota}$	$\frac{\iota}{3\sqrt[3]{\iota}}$
...	...	...	...	...
CT$_d$	$\sqrt[d]{\iota}$	d	$d\ell\sqrt[d]{\iota}$	$\frac{\iota}{d\sqrt[d]{\iota}}$
...	...	...	...	...
CT$_{\log_x m}$	x	$\log_x m$	$(\log_x m)\ell x$	$\frac{\iota}{x\log_x m}$

Table 5.1: Generalized CT results for single inheritance hierarchies

size of the representation and the dispatching time, and (ii) precise performance analysis, which dictates an optimal slice size, instead of the arbitrary universal recommendation, $x = 14$.

In reflecting on the generalized CT algorithm we see that they are readily adapted to the case where *message not understood* are allowed as is the case in dynamically typed languages. Whenever the search in a master-family F' returns $\top$, we can be certain that the search in every constituent of F' will also return $\top$. Therefore, it is possible to check after each dereferencing operation whether the fetched type is $\top$, and emit the appropriate error message. A more appealing alternative is to continue the search with $\top$, using an array which maps $\top$ into itself for each constituent of F'. Now, since this array does not depend on the identity of F', we can store only one such copy for each application of the CT reduction. The memory toll that CT$_d$ bears for these arrays is $(d-1)x$ cells.

Note also that Vitek and Horspool's idea of using selector coloring [44, 118] in each chunk is still applicable with a slight variation to our generalization. If certain columns in a chunk contain many $\top$ elements, it might be possible to collapse these columns together.

5.5 Incremental variants for Single Inheritance Hierarchies

This section describes an incremental variant of the CT scheme in the single inheritance setting, achieving two important properties: (i) the *space* it uses is at most twice that of the static algorithm, and (ii) its total *runtime* is linear in the final encoding size. (We cannot expect an asymptotically better runtime since the algorithm must at least output the final encoding.)

Section 5.5.1 describes ICT$_2$, the incremental variant of CT$_2$. Section 5.5.2 gives the generalization for CT$_d$.

The main idea is to *rebuild the entire encoding* whenever the ratio between the current slice size and the optimal one reaches a high- or low-water mark (for example 2 and 1/2). Therefore, some insertions will take longer to process than others. We therefore obtain bounds on the *amortized* time for an insertion.[2] The amortized time of an insertion is asymptotically optimal since the total runtime is linear in the final encoding size. Using techniques of “background copying” [42], it is possible to amend the algorithms so that the *worst case* insertion time is optimal as well.

Note that unlike the static version of the problem, we cannot assume that the families always include the root $\top$. The reason is that this assumption would require $\top$ to include implementation of *all* families, and the initial value of the number of families will jump to m.

5.5.1 Algorithm ICT$_2$ in a Single Inheritance Setting

The CT$_2$ scheme applies a single CT reduction and uses a dispatching matrix for the resulting master-families. This process divides the dispatching problem into independent sub-problems: one dispatching matrix, and a set of matrices A_i, $i = 1, \ldots, k$, which (in a single inheritance setting) are in fact dispatching matrices as well.

We first describe how to maintain a plain, single-level, dispatching matrix subject to type insertions. The insertion time will be linear in the encoding size, and the cost in dispatching time is in an additional comparison to guard against array overflows.

Each family is assigned a unique identifier in increasing order. The mapping of family-to-identifier is maintained as a hash-table. Consider a newly added type t. The newly introduced families[3] are assigned new identifiers and inserted into the hash-table. Observe that the dispatching result for such a newly introduced family and every other type is always null. However, instead of extending all the other rows with null entries, we perform a range-check before accessing any given row. In the case of array-overflow we return null, otherwise we proceed as usual.

The row of t in the dispatching matrix maps each family to its dispatching result. More precisely, the row of t is an extension of the row of its parent, except for entries corresponding to families in which t is a member. Note that the insertion time of a type is linear in its row size, and the total runtime is therefore linear in the final encoding size.

The space requirement of CT$_2$ in a single inheritance setting is (see Table 5.1)

$$\mathrm{mem}(x) = \ell x + nm/x, \tag{5.15}$$

which is minimized when the slice size is

$$x_{\text{OPT}} = \sqrt{nm/\ell}. \tag{5.16}$$

Algorithm ICT$_2$ will maintain the following invariant

$$\boxed{\frac{x_{\text{OPT}}}{2} \leq x \leq 2x_{\text{OPT}},} \tag{5.17}$$

and will rebuild the encoding whenever this condition is violated. Algorithm 5.1 shows the procedure to apply whenever a new type is added to the hierarchy.

[2]We remind the reader that the *amortized time* of an operation is $c(n)$, if a sequence of n such operations requires at most $nc(n)$ time. The worst case time of any single operation can however be much greater than $c(n)$. For more information on amortized complexity see [121].

[3]A new type t *introduces* a family F, $t \in F$, if and only if no other type was a member of F.

Algorithm 5.1 Insertion of a new type t in ICT$_2$

```
1: Let $x$ be the current slice size.
2: Let $\langle n, m, \ell \rangle$ be the current problem parameters.
3: $x_{\text{OPT}} \leftarrow \sqrt{nm/\ell}$        // The optimal slice size.
4: If not $\left(\frac{x_{\text{OPT}}}{2} \le x \le 2x_{\text{OPT}}\right)$ then
5:     $x \leftarrow x_{\text{OPT}}$
6:     Rebuild the entire CT$_2$ encoding
7: fi
8: Insert $t$ to the CT$_2$ encoding
```

Substituting (5.16) in (5.15) we find the optimal encoding size

$$\text{mem}(x_{\text{OPT}}) = 2\sqrt{nm\ell}.$$

Let us write this as a function of the problem parameters,

$$f(n, m, \ell) \equiv \text{mem}(x_{\text{OPT}}) = 2\sqrt{nm\ell}.$$

and study the properties of this function.

Fact 5.2 *Function f is monotonic in all three arguments n, m, ℓ.*

Fact 5.3 *There are constants c_1, c_2, c_3, such that*

$$\begin{aligned} \sum_{i=0}^{\infty} f\left(\frac{n}{2^i}, m, \ell\right) &\le c_1 f(n, m, \ell), \\ \sum_{i=0}^{\infty} f\left(n, \frac{m}{2^i}, \ell\right) &\le c_2 f(n, m, \ell), \\ \sum_{i=0}^{\infty} f\left(n, m, \frac{\ell}{2^i}\right) &\le c_3 f(n, m, \ell). \end{aligned} \tag{5.18}$$

PROOF. Note that

$$\begin{aligned} \sum_{i=0}^{\infty} f\left(\frac{n}{2^i}, m, \ell\right) &= \sum_{i=0}^{\infty} \sqrt{\frac{n}{2^i} m\ell} \\ &= \sqrt{nm\ell} \sum_{i=0}^{\infty} \sqrt{\frac{1}{2^i}} \\ &\le \frac{2}{2 - \sqrt{2}} \sqrt{nm\ell} \in O(f(n, m, \ell)). \end{aligned}$$

The proof for parameters m and ℓ is identical. □

Lemma 5.4 *The* space requirement *of* ICT$_2$ *is at most*

$$2f(n, m, \ell).$$

PROOF. From the algorithm invariant (5.17) it follows that

$$\begin{aligned}\mathrm{mem}(x) &= \ell x + nm/x \\ &\le \ell(2x_{\text{OPT}}) + nm/\left(\frac{x_{\text{OPT}}}{2}\right) \\ &= 2(\ell x_{\text{OPT}} + nm/x_{\text{OPT}}) \\ &= 2\,\mathrm{mem}(x_{\text{OPT}}) = 2f(n,m,\ell). \qquad \Box\end{aligned}$$

Our next objective is to prove that the total *runtime* of ICT$_2$ is linear in $f(n,m,\ell)$. To do so, we will breakdown the sequence of insertions carried out by the algorithm into *phases*, according to the points in time where rebuilding took place. No rebuilding occurs within a phase, and all that is required is to maintain several plain dispatching matrices. Hence, the total runtime of the insertions in a phase is linear in the encoding size at the end of this phase.

The main observation is that rebuilding happens only when at least one of the problem parameters is doubled. We distinguish between three *kinds* of rebuilds, depending on the parameter which was doubled. We then show that the total runtime of rebuilds of the same kind is linear in $f(n,m,\ell)$.

Formally, phase i begins immediately after phase $i-1$, and ends after the encoding was built for the i^{th} time (the last phase ends when the program terminates). Let $\langle n_i, m_i, \ell_i\rangle$, $i = 1,\ldots,p$, be the problem parameters at the end of phase i. Observe that the problem parameters can only increase, i.e., $n_{i+1} \ge n_i$, $m_{i+1} \ge m_i$, and $\ell_{i+1} \ge \ell_i$. Phase i finishes with an encoding size of at most $2f(n_i,m_i,\ell_i)$, therefore its runtime is linear in $f(n_i,m_i,\ell_i)$. Thus, the total runtime is linear in

$$\sum_{i=1}^{p} f(n_i,m_i,\ell_i). \tag{5.19}$$

We need to show that this sum is linear in $f(n_p,m_p,\ell_p)$.

Lemma 5.5 *Invariant* (5.17) *is violated only when* at least one *of the problem parameters is doubled, i.e., one of the following holds*

$$\begin{aligned} n_{i+1} &\ge 2n_i, \\ m_{i+1} &\ge 2m_i, \\ \ell_{i+1} &\ge 2\ell_i. \end{aligned} \tag{5.20}$$

PROOF. Let x_j denote the slice size at the beginning of phase j, i.e.,

$$x_j \equiv \sqrt{\frac{n_j m_j}{\ell_j}}. \tag{5.21}$$

At the end of phase i one of the following conditions must hold

$$\begin{aligned} x_{i+1} &\ge 2x_i, \\ x_{i+1} &\le \frac{1}{2}x_i. \end{aligned} \tag{5.22}$$

From (5.21) and (5.22), we have

$$\begin{aligned} \frac{n_{i+1}m_{i+1}}{\ell_{i+1}} &\geq \frac{4n_i m_i}{\ell_i}, \\ \frac{n_{i+1}m_{i+1}}{\ell_{i+1}} &\leq \frac{n_i m_i}{4\ell_i}. \end{aligned} \tag{5.23}$$

Since the problem parameters can only increase,

$$\begin{aligned} n_{i+1}m_{i+1} &\geq 4n_i m_i, \\ \ell_{i+1} &\geq 4\ell_i, \end{aligned} \tag{5.24}$$

which implies that at least one of the parameters was doubled. □

Lemma 5.6 *The total* runtime *of* ICT$_2$ *is linear in*

$$f(n_p, m_p, \ell_p).$$

PROOF. Let $\{(N_1, M_1, L_1), \ldots, (N_q, M_q, L_q)\}$ be the problem parameters of phases where n was doubled, i.e., $N_{i+1} \geq 2N_i$. Therefore,

$$N_q \geq 2N_{q-1} \geq \ldots \geq 2^{q-1}N_1. \tag{5.25}$$

Using Fact 5.3, the total runtime of these phases is linear in

$$\begin{aligned} \sum_{i=1}^{q} f(N_i, M_i, L_i) &\leq \sum_{i=1}^{q} f(N_i, M_q, L_q) \\ &\leq \sum_{i=1}^{q} f\left(\frac{N_q}{2^{q-j}}, M_q, L_q\right) \\ &\in O(f(N_q, M_q, L_q)). \end{aligned} \tag{5.26}$$

The same consideration applies to phases in which the number of methods or the number of families was doubled. So, the runtime of the entire algorithm is the total runtime of the three kinds of phases, which is linear in $f(n_p, m_p, \ell_p)$. □

5.5.2 Algorithm ICT$_d$ in a Single Inheritance Setting

The generalization to $d > 2$ is mostly technical, as outlined next. Function $\mathrm{mem}(x)$, the space requirement of CT$_d$ as defined in (5.14) is minimized when the slice size is

$$x_{\text{OPT}} = \sqrt[d]{nm/\ell}.$$

Let function f_d denote the optimal encoding size

$$f_d(n, m, \ell) \equiv \mathrm{mem}(x_{\text{OPT}}) = d\ell\sqrt[d]{\iota}.$$

Algorithm ICT$_d$ will preserve the following invariant

$$\boxed{\frac{x_{\text{OPT}}}{2^{1/(d-1)}} \leq x \leq 2x_{\text{OPT}}.} \tag{5.27}$$

Lemma 5.7 *The space requirement of* ICT_d *is at most* $2f_d(n, m, \ell)$.

PROOF. Similar to that of Lemma 5.4 □

Fact 5.8 *There are constants* c_1, c_2, c_3*, such that*

$$\begin{aligned}\sum_{i=0}^{\infty} f_d\left(\frac{n}{2^i}, m, \ell\right) &\leq c_1 f_d(n, m, \ell),\\ \sum_{i=0}^{\infty} f_d\left(n, \frac{m}{2^i}, \ell\right) &\leq c_2 f_d(n, m, \ell),\\ \sum_{i=0}^{\infty} f_d\left(n, m, \frac{\ell}{2^i}\right) &\leq c_3 f_d(n, m, \ell).\end{aligned} \tag{5.28}$$

Lemma 5.9 *Rebuilding only takes place when* at least one *of the problem parameters is doubled.*

PROOF. Similar to that of Lemma 5.5 □

Lemma 5.10 *The total runtime of* ICT_d *is linear in* $f_d(n_p, m_p, \ell_p)$.

PROOF. Similar to Lemma 5.6. □

5.6 Generalization of Compact Dispatch Tables for Multiple Inheritance Hierarchies

This section explains how to generalize the CT reduction as described in Section 5.4 to the multiple inheritance setting. In a single inheritance hierarchy, there could never be more than one most specific family member in response to a dispatch query. The fact that this is no longer true in multiple inheritance hierarchies makes it difficult to apply the CT reduction to such hierarchies. Even if the original families are appropriately augmented to remove all such ambiguities, ambiguities may still occur in the master-families as they are generated by the reduction.

We will therefore use a novel notion of a *generalized dispatching query*, denoted by $\textsf{g-dispatch}(F, t)$, which returns *the entire set* of smallest candidates, rather than null in case that this set is not a singleton. Formally,

$$\textsf{g-dispatch}(F, t) \equiv \min(\textsf{cand}(F, t)). \tag{5.29}$$

Generalized dispatching is a data-structure transaction rather than an actual runtime operation which must result in a single method to execute.

Consider for example the hierarchy of Figure 5.5.

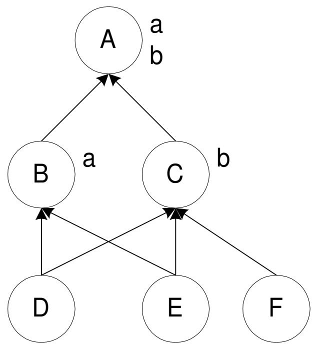


Figure 5.5: A small example of a multiple inheritance hierarchy with two families

The figure shows two families of methods, F_{a} and F_{b},

$$\begin{aligned} F_{\mathsf{a}} &= \{\mathsf{A}, \mathsf{B}\}, \\ F_{\mathsf{b}} &= \{\mathsf{A}, \mathsf{C}\}. \end{aligned} \tag{5.30}$$

The dispatching matrix of these two families is depicted in Figure 5.6a. Note that the results of all dispatching queries on types D and E (for example) are the same. The corresponding rows in the table are identical and can be compressed. Figure 5.6b shows a representation of the dispatching matrix obtained by merging together all identical rows and an auxiliary array of pointers to all different rows specimens.

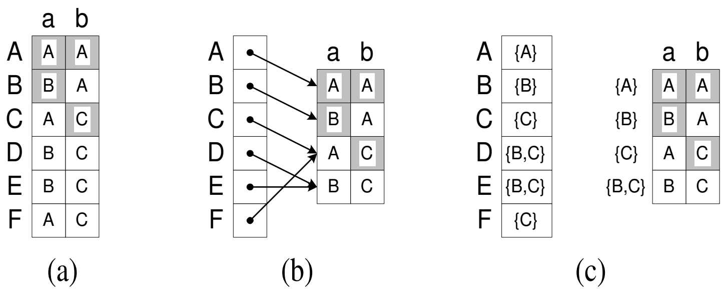


Figure 5.6: (a) The dispatching matrix of Figure 5.5, (b) the matrix compressed using an auxiliary array of pointers, and (c) the matrix compressed using an array of set-labels

This compressed representation can be understood in terms of the master-family

$$F' \equiv F_{\mathsf{a}} \cup F_{\mathsf{b}} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}.$$

The auxiliary array represents all the possible results of a *generalized* dispatch on this master-family. For example,

$$\textsf{g-dispatch}(F', \mathsf{D}) = \textsf{g-dispatch}(F', \mathsf{E}) = \{\mathsf{B}, \mathsf{C}\}.$$

Therefore, the D and E entries in the auxiliary array point to the same row specimen whose label is the set $\{\mathsf{B}, \mathsf{C}\}$.

In total there are four different results of generalized dispatching with respect to F'. Family F' therefore partitions the types in the hierarchy into four sets, as shown in Figure 5.6c. The figure shows the same compressed representation of the dispatching matrix, where the results of generalized dispatch are used to label row specimens instead of pointers in the auxiliary array.

In order to derive bounds on the quality of the CT compression in the multiple inheritance setting we need to estimate the number of distinct rows in chunks. The difficulty is that the result of a generalized dispatch is a set rather than a singleton, and hence this number might be exponential in the family size. To show that this is not the case, we first define the notion of a partition imposed by a family, and then show the size of this partition is at most 2κ times the size of the family, where $1 \leq \kappa \leq n$ is a (usually small) metric of the complexity of the hierarchy.

5.6.1 Family Partitionings

Given a partially ordered set of types $\mathcal{T}$ and a family of implementations $F \subseteq \mathcal{T}$, the *partitioning of $\mathcal{T}$ by F*, also called the *family partitioning* due to F, is

$$\nabla F \equiv \{\mathcal{T}_1, \ldots, \mathcal{T}_n\},$$

such that all types in *partition* $\mathcal{T}_i$ have the same generalized dispatch result. In other words, types $a, b \in \mathcal{T}$ are in the same *partition* $\mathcal{T}_i \in \nabla F$ if and only if

$$\textsf{g-dispatch}(F, a) = \textsf{g-dispatch}(F, b). \tag{5.31}$$

Figure 5.7 shows the family partitioning of the families $F_{\textsf{a}}$, $F_{\textsf{b}}$ of (5.30) and their master-family $F' \equiv F_{\textsf{a}} \cup F_{\textsf{b}}$.

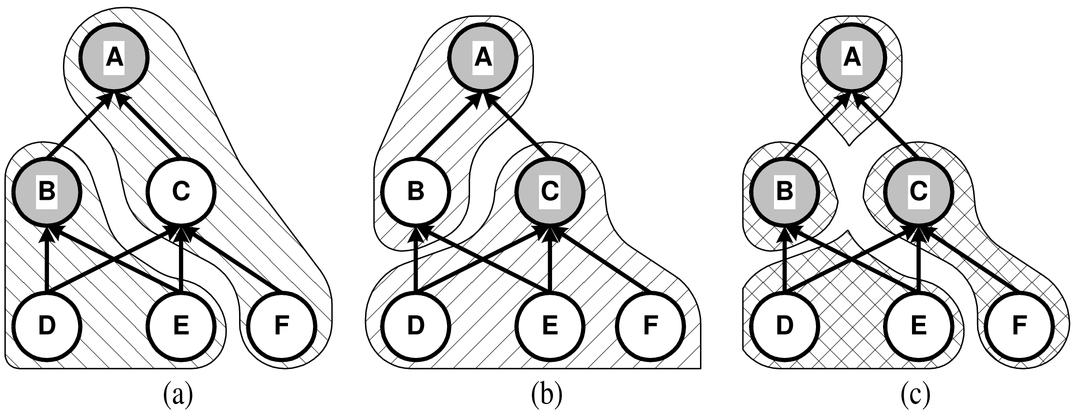


Figure 5.7: The family partitionings of the families $F_{\textsf{a}}$, $F_{\textsf{b}}$ of (5.30) and their master-family $F' \equiv F_{\textsf{a}} \cup F_{\textsf{b}}$

Types D and E, for example, are in the same partition in $\nabla F'$ since $\textsf{g-dispatch}(F', \textsf{D}) = \textsf{g-dispatch}(F', \textsf{E}) = \{\textsf{B}, \textsf{C}\}$. The partitionings are

$$\begin{aligned}
\nabla F_{\textsf{a}} &\equiv \{\{\textsf{A}, \textsf{C}, \textsf{F}\}, \{\textsf{B}, \textsf{D}, \textsf{E}\}\}, \\
\nabla F_{\textsf{b}} &\equiv \{\{\textsf{A}, \textsf{B}\}, \{\textsf{C}, \textsf{D}, \textsf{E}, \textsf{F}\}\}, \\
\nabla F' &\equiv \{\{\textsf{A}\}, \{\textsf{B}\}, \{\textsf{C}, \textsf{F}\}, \{\textsf{D}, \textsf{E}\}\}.
\end{aligned} \tag{5.32}$$

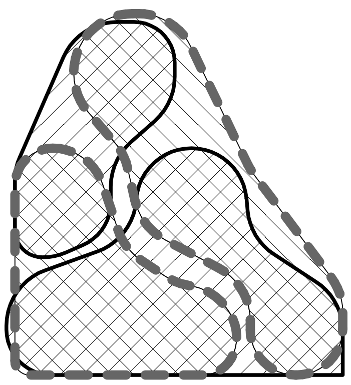


Figure 5.8: The *overlay* of ∇F_{a} and ∇F_{b} of Figure 5.7

Figure 5.8 overlays ∇F_{a} and ∇F_{b}. The dotted lines are the partitions of ∇F_{a}, whereas the full lines are the partitions of ∇F_{b}.

In comparing Figure 5.7c with Figure 5.8, we see that the partitioning $\nabla F'$ can be obtained by a simple overlay of the two partitionings ∇F_{a} and ∇F_{b}. We will next prove that this was no coincidence.

Given two partitionings π, π', their *overlay* $\pi \cdot \pi'$ is the coarsest partitioning consistent with both π and π'. Constructively, the overlay is obtained by intersecting all partitions of π with all partitions of π':

$$\pi \cdot \pi' = \{\mathcal{T}_i \cap \mathcal{T}'_j \mid \mathcal{T}_i \in \pi, \mathcal{T}'_j \in \pi'\}. \tag{5.33}$$

For example, the overlay of ∇F_{a} and ∇F_{b} of (5.32) is

$$\begin{aligned}\nabla F_{\mathsf{a}} \cdot \nabla F_{\mathsf{b}} &= \{\{\mathsf{A},\mathsf{C},\mathsf{F}\} \cap \{\mathsf{A},\mathsf{B}\}, \{\mathsf{A},\mathsf{C},\mathsf{F}\} \cap \{\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{F}\},\\ &\qquad \{\mathsf{B},\mathsf{D},\mathsf{E}\} \cap \{\mathsf{A},\mathsf{B}\}, \{\mathsf{B},\mathsf{D},\mathsf{E}\} \cap \{\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{F}\}\}\\ &= \{\{\mathsf{A}\}, \{\mathsf{C},\mathsf{F}\}, \{\mathsf{B}\}, \{\mathsf{D},\mathsf{E}\}\}.\end{aligned} \tag{5.34}$$

Lemma 5.11 $\nabla F_1 \cdot \nabla F_2 = \nabla(F_1 \cup F_2)$ *for all* F_1, F_2.

PROOF. It is a well known fact that for every partitioning π there is a binary *equivalence relation* whose set of equivalence classes are the same as the partitioning π. Instead of proving that the partitioning $\nabla(F_1 \cup F_2)$ and $\nabla F_1 \cdot \nabla F_2$ are equal, we will prove that their equivalence relations are the same.

On the one hand, types a, b are in the equivalence relation of

$$\nabla(F_1 \cup F_2)$$

if and only if they have the same generalized dispatching results with respect to $F_1 \cup F_2$ (see (5.31)), i.e.,

$$\textsf{g-dispatch}(F_1 \cup F_2, a) = \textsf{g-dispatch}(F_1 \cup F_2, b). \tag{5.35}$$

On the other hand, the overlay partitioning, $\nabla F_1 \cdot \nabla F_2$, is defined by intersecting all partitions of ∇F_1 with those of ∇F_2 (see (5.33)). Therefore, types a, b are in the equivalence relation of $\nabla F_1 \cdot \nabla F_2$ iff the following two conditions hold

$$\begin{aligned} &\textsf{g-dispatch}(F_1, a) = \textsf{g-dispatch}(F_1, b), \\ &\textsf{g-dispatch}(F_2, a) = \textsf{g-dispatch}(F_2, b). \end{aligned} \tag{5.36}$$

We must show that (5.35) holds iff (5.36) holds. Formally, using the definition of generalized dispatch (5.29), we must show that

$$\begin{aligned} &\min(\textsf{cand}(F_1 \cup F_2, a)) = \min(\textsf{cand}(F_1 \cup F_2, b)) \\ &\qquad\qquad \Leftrightarrow \\ &\quad \min(\textsf{cand}(F_1, a)) = \min(\textsf{cand}(F_1, b)) \ \wedge \\ &\quad \min(\textsf{cand}(F_2, a)) = \min(\textsf{cand}(F_2, b)). \end{aligned} \tag{5.37}$$

Since two sets of candidates (for the same family) have the same smallest elements if and only if they are equal, our objective is to prove (see the definition of candidates in (5.1))

$$\begin{aligned} &(F_1 \cup F_2) \cap \text{ancestors}(a) = (F_1 \cup F_2) \cap \text{ancestors}(b) \\ &\qquad\qquad \Leftrightarrow \\ &\quad F_1 \cap \text{ancestors}(a) = F_1 \cap \text{ancestors}(b) \ \wedge \\ &\quad F_2 \cap \text{ancestors}(a) = F_2 \cap \text{ancestors}(b). \end{aligned} \tag{5.38}$$

Given two sets X, Y, their *symmetric difference* is defined as

$$X \bigtriangleup Y \equiv (X \cup Y) \setminus (X \cap Y).$$

Observe that

$$Z \cap X = Z \cap Y \Leftrightarrow Z \cap (X \bigtriangleup Y) = \emptyset. \tag{5.39}$$

By combining (5.38) and (5.39) we find that we need to prove that

$$\begin{aligned} &(F_1 \cup F_2) \cap (\text{ancestors}(a) \bigtriangleup \text{ancestors}(b)) = \emptyset \\ &\qquad\qquad \Leftrightarrow \\ &\quad F_1 \cap (\text{ancestors}(a) \bigtriangleup \text{ancestors}(b)) = \emptyset \ \wedge \\ &\quad F_2 \cap (\text{ancestors}(a) \bigtriangleup \text{ancestors}(b)) = \emptyset. \end{aligned} \tag{5.40}$$

The above trivially holds since for all sets X, Y, Z,

$$\begin{aligned} &(X \cup Y) \cap Z = \emptyset \\ &\qquad \Leftrightarrow \\ &\quad X \cap Z = \emptyset \ \wedge \\ &\quad Y \cap Z = \emptyset. \qquad\qquad \square \end{aligned}$$

5.6.2 Memory Requirements of the Reduction

As in the single inheritance version, the CT reduction partitions the families $\mathcal{F}$ into disjoint slices $\mathcal{F}_1, \ldots, \mathcal{F}_k$, and generates for the i^{th} slice the master-family F'_i by merging the families in this slice. To answer the *generalized dispatching* query g-dispatch(F, t), where $F \in \mathcal{F}_i$, we first (recursively) answer the query g-dispatch(F'_i, t), in the collection of master-families, $\{F'_1, \ldots, F'_k\}$. This recursive call returns one of the partitions of $\nabla F'_i$. The next step is to find the unique containing partition of ∇F.

To understand this better, recall that $F \subseteq F'_i$. To apply Lemma 5.11 note that there exists a set X such that $F'_i = F \cup X$, and hence

$$\nabla F'_i = \nabla(F \cup X) = \nabla F \cdot \nabla X.$$

Therefore, every partition of $\nabla F'_i$ is contained in a partition of ∇F. A matrix A_i with $|\nabla F'_i|$ rows and $|\mathcal{F}_i|$ columns is used to map each of the partitions of $\nabla F'_i$ to a partition of ∇F, for all $F \in \mathcal{F}_i$. Matrices $A_1, \ldots, A_k$ are nothing other than the dispatching data structure of the CT reduction. (Clearly, there is an additional data structure which the recursive call uses.)

To bound the size of these matrices, we need to bound $|\nabla F|$. In single inheritance, the root of each partition correspond to a different family member, and therefore $|\nabla F| \leq |F|$. An easy, but not so useful bound in multiple inheritance, is $|\nabla F| \leq 2^{|F|}$.

A better bound is given by defining κ, the complexity of a hierarchy, and then showing that

$$|\nabla F| \leq 2\kappa|F|. \tag{5.41}$$

Using slices with x families in each, the total memory of matrices $A_1, \ldots, A_k$ is

$$\sum_{i=1}^{k} |\nabla F'_i| \times |\mathcal{F}_i| = \sum_{i=1}^{k} |\nabla F'_i| \times x \leq x \sum_{i=1}^{k} 2\kappa |F'_i| \leq 2x\kappa\ell.$$

The recursive equations then become

$$\begin{aligned} \mathrm{mem}_1(n, m, \ell) &= nm, \\ \mathrm{mem}_{d+1}(n, m, \ell) &\leq 2\kappa\ell \cdot x + \mathrm{mem}_d(n, m/x, \ell). \end{aligned} \tag{5.42}$$

By using $2\kappa\ell$ instead of ℓ, the analysis of the previous section holds.

Corollary 5.12 *Let $\varphi \equiv (nm)/(2\kappa\ell)$. In a hierarchy whose complexity is κ,* CT$_d$ *performs dispatching in d dereferencing operations, and reaches a compression factor of at least $\frac{1}{d}\varphi^{1-1/d}$ (when using a slice size of $\varphi^{1/d}$).*

In other words, in a hierarchy whose complexity is κ, the space requirements of CT$_d$ in the multiple inheritance setting is worse than the single inheritance setting by a factor of at most $(2\kappa)^{1-1/d}$.

5.6.3 Hierarchy Complexity

Definition 5.13 *The complexity of a hierarchy is the minimal number κ such that there exists partitioning of $\mathcal{T}$ into sets $\mathcal{T}_1, \ldots, \mathcal{T}_\kappa$, and an ordering π_i of $\mathcal{T}_i$, $i = 1, \ldots, \kappa$, such that for every type $t \in \mathcal{T}$, the set* $\mathrm{descendants}(t) \cap \mathcal{T}_i$ *is an interval in π_i.*

Clearly, the complexity of a hierarchy is 1 if there exists an ordering π of $\mathcal{T}$ in which descendants of any type define an *interval*. All single inheritance hierarchies have complexity 1 since in a simple preorder the descendants of any type are consecutive.

Figure 5.9 is a multiple inheritance hierarchy of complexity 1. Within each type we write its position in π.

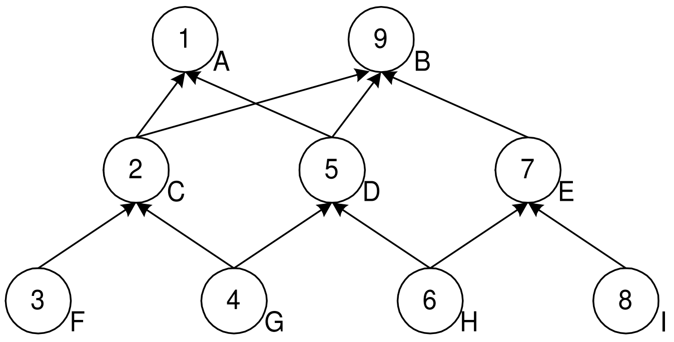


Figure 5.9: An example of a multiple inheritance hierarchy of complexity 1

Figure 5.10 shows the family partitioning of the family $F = \{\mathsf{A}, \mathsf{B}, \mathsf{E}\}$ in the hierarchy of Figure 5.9. Observe that $|\nabla F| = 5$.

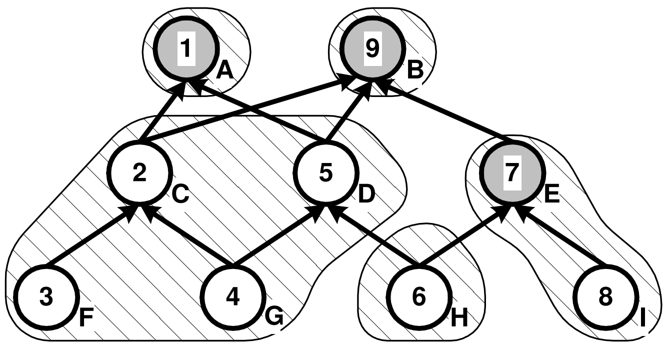


Figure 5.10: The family partitioning of the family $F = \{\mathsf{A}, \mathsf{B}, \mathsf{E}\}$ in the hierarchy of Figure 5.9

Since the complexity of this hierarchy is 1, the descendants of each type define an *interval*. Therefore the family F defines the three intervals depicted in Figure 5.11.

The intervals in Figure 5.11 partition the types into 5 *segments*. (We will show that there are at most $2|F|$ segments.) Types in the same segment have the same set of candidates and therefore belong to the same partition. So we conclude that the number of

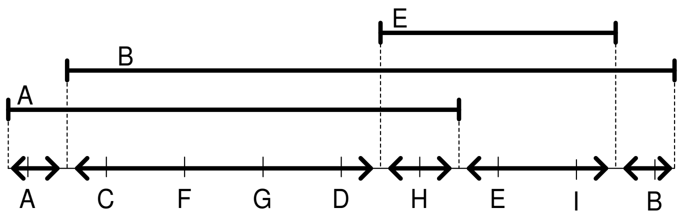


Figure 5.11: The intervals of the family $F = \{\mathsf{A}, \mathsf{B}, \mathsf{E}\}$ in the hierarchy of Figure 5.9

partitions is at most the number of segments, which in turn is at most $2|F|$. In our example,

$$|\nabla F| = 5 \leq 6 = 2|F|.$$

Lemma 5.14 $|\nabla F| \leq 2\kappa|F|$ *for each family* F.

PROOF. We need the following fact, whose proof is elementary.

> *A set of f intervals partition any consecutive set into at most $2f+1$ segments. Out of these segments at most $2f-1$ are contained in one interval or more.* (See illustration in Figure 5.11.)

Let $f = |F|$. Recall (Definition 5.13) the partitioning of $\mathcal{T}$ into sets $\mathcal{T}_1, \ldots, \mathcal{T}_\kappa$ with their respective ordering. Let i be fixed. We write the list of members of the set $\mathcal{T}_i$, enumerated in its respective order π_i.

Consider a type $t \in \mathcal{T}_i$. The result of $\mathsf{g\text{-}dispatch}(F, t)$ is uniquely determined by the subset of all types $t' \in F$, such that the t is among the descendant of t'. From Definition 5.13, we have that the descendants are consecutive in the list of $\mathcal{T}_i$. Family F defines therefore f intervals (which may be empty) in this list. These intervals partition the list into at most $2f + 1$ segments such that the result of $\mathsf{g\text{-}dispatch}(F, t)$ is uniquely determined by the segment of t. These segments give the restriction of ∇F to $\mathcal{T}_i$.

We have thus obtained $|\nabla F| \leq \kappa(2f + 1)$. To obtain a tighter bound we need a more careful counting. Let us remove from $\mathcal{T}_i$ all types which are not descendants of any of the members of F. The remaining types are divided by F into $2f - 1$ segments. Generalized dispatching on the removed types returns the empty set, irrespective of i. The total number of equivalence classes in ∇F is therefore $\kappa(2f - 1) + 1 \leq 2\kappa f$. $\square$

We are unaware of any non-exponential method for finding κ. Instead we use the PQ-trees heuristic [137] which gives an *upper-bound* on κ. On a benchmark of 19 large multiple inheritance hierarchies, the median value on that bound was 5, the average was 6.4, and the maximum was 18.

Remark 5.15 *The actual partitioning $\mathcal{T}_1, \ldots, \mathcal{T}_\kappa$ is* not *required in order to apply the CT reduction; only the integer κ is needed for determining the slice size. We found that in practice the single inheritance analysis closely models even hierarchies which use multiple inheritance heavily. (Therefore there is no need even to find κ.)*

5.7 Experimental Results

In this section we compare the theoretical prediction on the algorithms with their empirical performance. Our benchmark comprises 35 hierarchies totaling 63,972 types, 70,680 messages and 418,839 methods. Out of these, there were 16 single inheritance hierarchies with 29,162 types, 12 multiple inheritance hierarchies with 27,728 types, and 7 multiple dispatch hierarchies with 7,082 types.

This data-set includes all hierarchies previously used in the literature in benchmarks of dispatching algorithms. However, prior to running the experiments, all degenerate families, i.e., families of size one, were pruned from the input. The reason for doing so is that sending a message whose family is degenerate requires no dispatching, and is the same as static procedure call. (In dynamically typed languages there is an earlier step, which is *equivalent* to a subtyping test, in which it is made sure that the message is valid for the receiver type.)

We stress that by eliminating degenerate families, compression becomes *more difficult* for the CT schemes. The reason is that this pruning reduces both m and ℓ by the same number. Therefore, the optimal compression factor $\iota \equiv (nm)/\ell$, which we aimed at reaching, becomes smaller. On the other hand, the compression factor of null-*elimination schemes* $(nm)/w$ may or may not decrease.

Table 5.2 gives the essential properties of the pruned hierarchies. The first two row blocks in the table correspond to single inheritance (SI) and multiple inheritance (MI) hierarchies. The last block is for hierarchies drawn from multi-dispatch languages. (We regard each multi-dispatch query as several independent single-dispatch queries on each of the arguments, as done in the first step of the major algorithms for multi-dispatching [137].)

The first two data columns in the table give the values of n and m for each of the hierarchies in the data-set. We see that the hierarchies span a range of sizes: the number of types is between 107 and 8,793 while the number of messages is between 131 and 14,575. A more detailed description of the data-set, including the source of the hierarchies and their respective programming languages is available elsewhere [137].

The column entitled $\frac{nm}{10^6}$ gives the memory requirement of the dispatching matrix, measured in millions of cells. We see that this matrix can be huge. Suppose that each cell uses four bytes (an assumption we make henceforth), then this matrix consumes about 160MB of memory in the MI: JDK 1.3.1 hierarchy and about 500MB in the MI: IBM SF hierarchy.

The next column in the table, entitled $\frac{w}{10^3}$, gives the number of non-null entries in the dispatching matrix, measured in thousands. The column indicates that this matrix is sparse: In most cases, 90% or more of its cells are empty. We shall use this column as a *baseline* for comparison of the CT algorithms, since it shows the memory requirement of an *optimal* null-elimination scheme such as VFT on single inheritance hierarchies. Note that in hierarchies such as MI: JDK 1.3.1 and MI: IBM SF the potential compression is by a factor of 300 or more. But still, the VFTs may consume a lot of space: 1–2MB on some single inheritance hierarchies.

The column entitled $\frac{\ell}{10^3}$ gives the number of method implementations, which ranges

	Hierarchy	n	m	$\frac{(nm)}{10^6}$	$\frac{w}{10^3}$	$\frac{\ell}{10^3}$	$\overline{\kappa}$
Single Inheritance	Visualworks1	774	1,170	0.91	79.14	4.62	1
	Visualworks2	1,956	3,196	6.25	289.67	13.58	1
	Digitalk2	535	962	0.51	72.27	3.33	1
	Digitalk3	1,357	2,402	3.26	362.11	9.44	1
	IBM Smalltalk 2	2,320	4,335	10.06	204.97	16.29	1
	VisualAge 2	3,241	6,529	21.16	594.98	26.21	1
	NextStep	311	499	0.16	16.24	2.12	1
	ET++	371	296	0.11	12.20	1.41	1
	SI: JDK 1.3.1	6,681	4,392	29.34	128.26	23.82	1
	SI: Corba	1,329	222	0.30	6.94	2.59	1
	SI: HotJava	644	690	0.44	23.86	2.91	1
	SI: IBM SF	6,626	11,664	77.29	287.38	88.28	1
	SI: IBM XML	107	131	0.01	1.30	0.59	1
	SI: Orbacus	1,053	980	1.03	18.66	3.82	1
	SI: Orbacus Test	579	368	0.21	5.67	2.39	1
	SI: Orbix	1,278	535	0.68	10.90	2.90	1
Multiple Inheritance	Self	1,802	2,459	4.43	234.04	21.75	3
	Unidraw	614	360	0.22	8.11	2.33	2
	LOV	436	663	0.29	14.09	2.84	11
	Geode	1,318	1,413	1.86	122.27	9.52	18
	MI: JDK 1.3.1	7,401	5,724	42.36	140.91	28.68	9
	MI: Corba	1,699	396	0.67	13.58	3.20	7
	MI: HotJava	736	829	0.61	24.90	3.40	7
	MI: IBM SF	8,793	14,575	128.16	390.35	116.15	12
	MI: IBM XML	145	271	0.04	2.33	0.95	3
	MI: Orbacus	1,379	1,261	1.74	24.82	5.00	4
	MI: Orbacus Test	689	379	0.26	7.49	2.75	4
	MI: Orbix	2,716	786	2.13	22.44	3.70	4
Multiple Dispatching	Cecil	932	1,009	0.94	72.89	4.21	5
	Dylan	925	428	0.40	70.38	1.78	3
	Cecil-	473	592	0.28	16.06	2.36	5
	Cecil2	472	131	0.06	17.17	0.56	5
	Harlequin	666	229	0.15	23.11	1.02	8
	Vor3	1,660	328	0.54	15.44	1.86	5
	Vortex3	1,954	476	0.93	305.50	2.50	7

Table 5.2: Essential parameters of the pruned hierarchies in our data-set

between 562 and 116,152. This column also sets a lower bound on the memory used by an *optimal* duplicates-elimination compression scheme. Comparing this column to the previous one, we learn that duplicates-elimination is potentially much better than null-elimination. However, it is much more difficult to come close to optimal duplicates-elimination than to optimal null-elimination. We shall use this column as another comparison standard for the performance of the CT algorithms.

The final column entitled $\overline{\kappa}$ shows an upper bound on κ which was found by our PQ-trees[4] heuristic [137]. (Recall that we do not have an efficient algorithm for computing κ.)

[4]We used the slow PQ-trees heuristic instead of the fast order-preserving heuristic in order to obtain a

In single inheritance hierarchies, $\kappa = \overline{\kappa} = 1$. The median of $\overline{\kappa}$ in the remaining hierarchies is 5. The hierarchy whose topology seems to be the most complex is Geode, followed by MI: IBM SF, LOV and then JDK 1.3.1.

The implementation of the various CT schemes was run on 900Mhz Pentium III computer, equipped with 256MB internal memory and controlled by a Windows 2000 operating system. On this machine, the runtimes for generating the encoding (without actually copying the values into matrices) of the first four schemes (CT_2 through CT_5) were 0.7 Sec, 1.4 Sec, 2.1 Sec and 2.9 Sec. Since our data-set included in total 418,839 methods we find that the time per implementation is measured in microseconds. For example, we found that the creation time *per implementation* ranged between 0.3 and 1.7 μSec in CT_2 in single inheritance hierarchies (the median being 0.6 μSec). These times increase in multiple inheritance hierarchies: the range being 1.1 to 6.7 μSec; the median being 2.4 μSec.

Figure 5.12 shows the memory used by the first four CT schemes relative to the w baseline in the 35 hierarchies in the data-set. Memory usage of the CT schemes were obtained using the empirically found *best* slice size (which may be different than the prescription of column 2 of Table 5.1).

The figure shows that compared to the *optimal* null-elimination, CT_2 is better in 6 hierarchies, CT_3 in 13 hierarchies, CT_4 in 15 hierarchies, and CT_5 in 16 hierarchies. In a few cases, the improvement is by an order of magnitude from the baseline. We also see that CT_2 is at most one order of magnitude worse than this idealized baseline.

We can also learn from Figure 5.12 that the incremental improvement by the series of CT schemes is diminishing. In fact, examining the actual memory requirements, we find that the median incremental improvements are: CT_3 over CT_2: 44%, CT_4 over CT_3: 18%, and CT_5 over CT_4: 8%. This finding is consistent with the theoretical prediction.

The figure also plots another idealized algorithm, i.e., the optimal duplicates-elimination scheme, which uses ℓ cells. We see that this ideal is about one order of magnitude better than the various CT schemes. Finally, we see a certain correlation between ℓ and the series of CT schemes, as predicted by the theoretical analysis. When $\ell \ll w$ the CT schemes outperform even an optimal null-elimination scheme.

We now turn to comparing the actual performance of the various CT schemes with the theoretically obtained bounds.

In single inheritance hierarchies, the upper bound on the memory requirement are given by the fourth column of Table 5.1. Figure 5.13a shows the memory requirement relative to these values. We see that in all schemes and in all hierarchies, the memory requirement is significantly smaller than the upper bounds. Also, the extent of improvement of CT_d over the upper bound increases with d.

Corollary 5.12 provides the upper bounds in multiple inheritance hierarchies depending on their complexity κ. Figure 5.13b shows the memory, relative to these upper bounds, of the actual CT performance. Again, we see that the extent of improvement of CT_d over the upper bound increases with d. Interestingly, in comparing Figure 5.13b with Fig-

better upper bound. We will see that, in practice, it is better to use $\kappa = 1$ to find the slice size, so the speed of the heuristic is irrelevant.

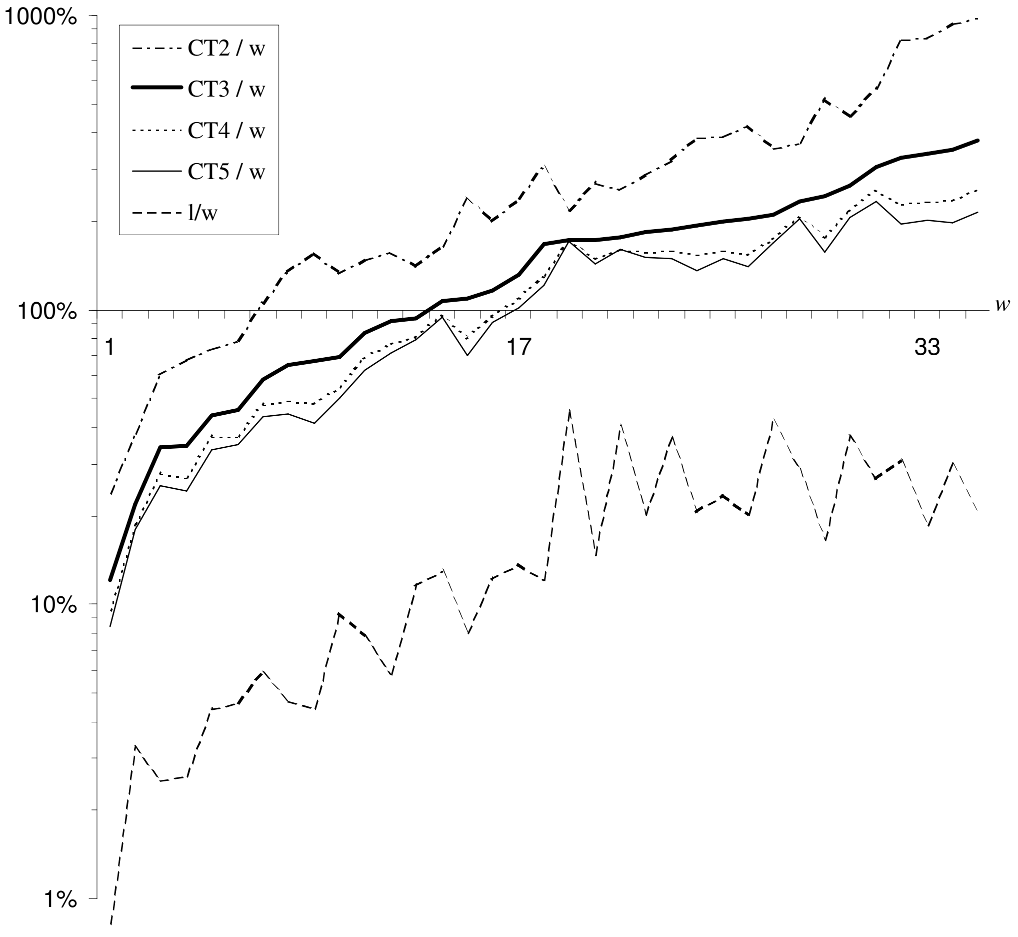


Figure 5.12: Memory used by CT_2, CT_3, CT_4, CT_5 and optimal duplicates-elimination (ℓ) relative to optimal null-elimination (w – marked as the 100%); hierarchies are sorted in ascending memory used by CT_3

ure 5.13a, we see that the improvement of the implementation upon the upper bound is much greater in multiple inheritance vs. single inheritance hierarchies.

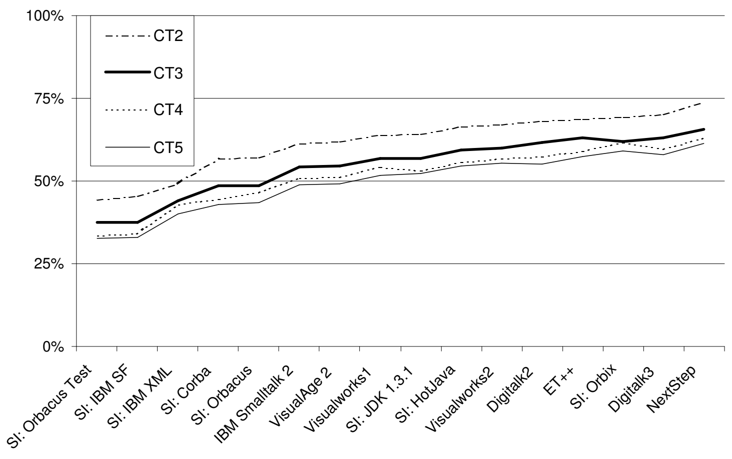
A possible explanation for this seemingly better performance in multiple inheritance hierarchies is exaggerated upper bounds. Examining Corollary 5.12, we see that the upper bounds increase with κ. Since our heuristics only finds an upper approximation of κ, it could be that the true upper bounds are actually smaller, and hence the improvement upon the upper bound is not as great.

Figure 5.13c tries to test this hypothesis, by comparing the performance on multiple inheritance hierarchies with the upper bounds obtained by assuming $\kappa = 1$ (as in single inheritance hierarchies).[5] We see that the improvement upon the upper bounds computed thus is almost the same as in single inheritance hierarchies (Figure 5.13a). Such a similarity could not be explained by an overestimation of κ.

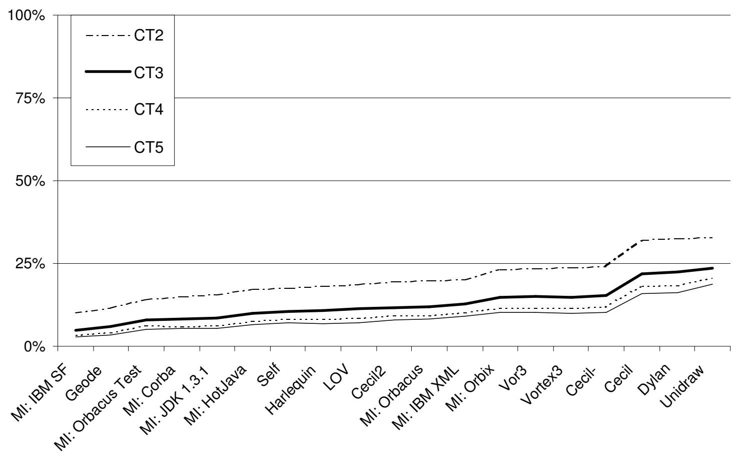
The reason that the CT algorithms perform better than the theoretically obtained

[5]In fact, we used the bound for single inheritance in Table 5.1, which is smaller by a factor of $(2\kappa)^{1-1/d}$ than the bound for multiple inheritance in Corollary 5.12.

(a)



(b)



(c)

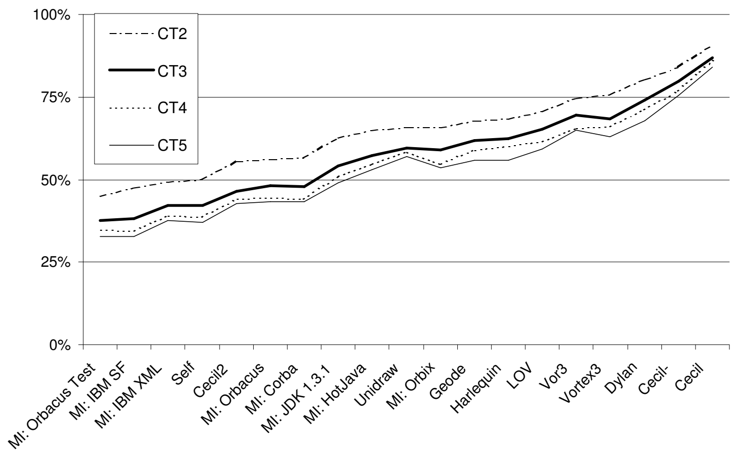


Figure 5.13: The memory requirement of CT_2, CT_3, CT_4 and CT_5 relative to the theoretically obtained upper bounds in single inheritance hierarchies (a), multiple inheritance hierarchies where the upper bound was computed using $\overline{\kappa}$ (b), and multiple inheritance hierarchies, where the upper bound is computed as in single inheritance hierarchies ($\kappa = 1$) (c)

bounds is that the analysis of the CT reduction bounded the size of a master-family by the sum of sizes of its constituents, i.e.,

$$|F_i'| = \left| \bigcup_{F \in \mathcal{F}_i} F \right| \leq \sum_{F \in \mathcal{F}_i} |F|.$$

In fact, especially when the families are large, the probability of finding shared elements may be significant, and the master family is likely to be smaller. As a result, ℓ', the number of implementations after the reduction, may be much smaller than the original value ℓ. For example, with $x = 29$ for CT$_2$ in Digitalk3, the CT reduction transforms the problem $\langle n, m, \ell \rangle = \langle 1357, 2402, 9444 \rangle$ to $\langle 1357, 83, 4616 \rangle$, i.e., the number of implementations decreased by a factor of more than 2. Our analysis assumed (see (5.11)) however that $\ell' = \ell$.

This effect increases also with slice size, which is the reason that choosing a slice size greater than the theoretical prescription may improve the performance of the reduction. In IBM SF, for example, the theoretical analysis suggested that $x_{\text{OPT}} = 30$ as optimal slice size for CT$_2$. However, by using instead a slice size $x = 70$, we were able to further reduce the number of cells from 3.3M to about 2.4M.

Figure 5.14 compares the actual memory used by the CT$_2$ scheme with the theoretical prediction (5.14) in the Digitalk3 hierarchy. (The graphs of other hierarchies and higher order schemes are similar.) We see that the extent by which the empirical performance is superior to the theoretically obtained bound increases with the slice size.

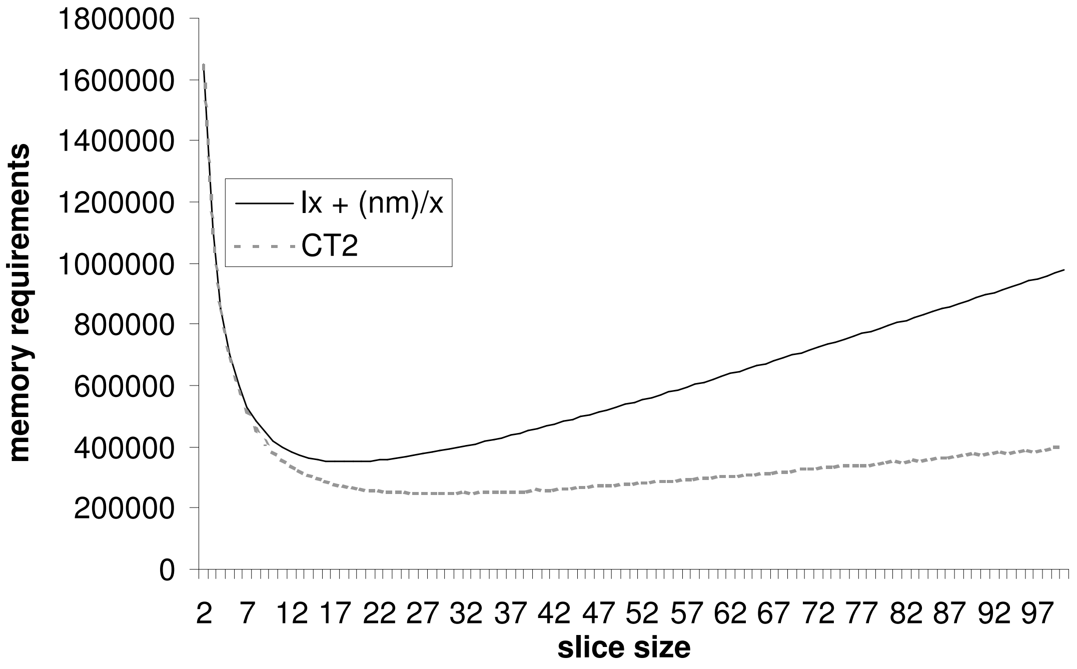


Figure 5.14: Space requirements vs. slice size in the single inheritance hierarchy of Digitalk3 for CT$_2$ and its theoretical upper bound (5.14)

5.8 Conclusions and Open Problems

The incremental algorithm described in Section 5.5 is in many ways the pinnacle of this chapter. This algorithm assumes the single inheritance, dynamically typed, and dynamic loading model, denoted SDTDL. A prime example for the model is the SMALLTALK programming language. Note that the VFT method is unsuitable in an SDTDL model.

Curiously, even though JAVA is in essence a statically typed language, the implementation of the `invokeinterface` bytecode instruction is a very close match of this model. To see this recall that all implementations of a method defined in an **`interface`** must reside in **`class`**es, and that these classes take a tree topology. The locations of these implementations in this tree are however totally unrelated, and additional implementations can be introduced as a result of dynamic class loading. Even though there is a possibility of using static information of the interface type, many implementations of the `invokeinterface` bytecode instruction assume an SDTDL model.

Incorporating the algorithm into a runtime system requires careful attention to details, including selecting a heuristic of determining the optimal slice size, which might perform better than the theoretical value, a wise strategy for background copy to avoid stagnation, tweaking and fine tuning of the partitioning algorithm, etc. We leave this empirical evaluation to continuing research.

Also, the incremental algorithm can be generalized to the multiple inheritance setting, but there are subtle issues in the theoretical analysis of the performance of this generalization.

Observe that the static algorithm for multiple inheritance hierarchies, achieves $2\kappa\ell \lg m$ space when $d = \lg m$. Type slicing [137] however uses only $O(\kappa\ell)$ cells, while achieving $O(\lg\lg n)$ dispatching time. There is therefore a reason to believe that the tradeoff offered by our technique can be improved, especially for higher values of d.

Chapter 6

Two-Dimensional Bi-Directional Object Layout

Chapter Summary
C++ object layout schemes rely on (sometimes numerous) compiler generated fields. We describe a new language-independent object layout scheme, which is space optimal, i.e., objects are contiguous, and contain no compiler generated fields *other than a single type identifier. As in C++ and other multiple inheritance languages such as Cecil and Dylan, the new scheme sometimes requires extra levels of indirection to access some of the fields. Using a data set of 28 hierarchies, totaling almost 50,000 types, we show that the new scheme improves field access efficiency over standard implementations, and competes favorably with (the non-space optimal) highly optimized C++ specific implementations. The benchmark includes a new analytical model for computing the frequency of indirections in a sequence of field access operations. Our layout scheme relies on whole-program analysis, which requires about 10 micro-seconds per type on a contemporary architecture (Pentium III, 900Mhz, 256MB machine), even in very large hierarchies.*

A common argument raised by proponents of the single inheritance programming model is that multiple inheritance incurs space and time overheads and inefficiencies on the runtime system [24,94]. A large body of research was targeted at reducing the multiple inheritance overhead in operations such as dynamic message dispatch and subtyping tests (see e.g., [136–138] for recent surveys). Another great concern in the design of runtime systems for multiple inheritance hierarchies is efficient object layout. To this end, both general purpose [113] and C++ language specific [53, 68] object layout schemes were previously proposed in the literature.

The various C++ layout schemes are not space-optimal since they introduce (sometimes many) compiler generated fields into the layout. They are also not time-optimal since access to certain fields (in particular, those defined in virtual bases) requires several memory dereferences. This thesis revisits the object layout problem in the general, language-independent setting. Our new object layout scheme is space optimal, i.e., objects are contiguous, and contain *no compiler generated fields*. Hence, in terms of space, it is superior to C++ layout schemes. It is also superior in terms of field access efficiency to the space-optimal *field dispatching* technique[1] employed by many object oriented languages.

We say that the layout is *two dimensional, bi-directional* since all objects can be thought of as being laid out first in a two-dimensional matrix, whose rows (also called *layers*) may span both positive and negative indices. The layout algorithm ensures that the populated portion of each such layer is consecutive, regardless of the object type. The particular object layout in one-dimensional memory is a cascade of these portions.

A data set of 28 hierarchies, totaling almost 50,000 types, was used in comparing the field access efficiency of the new scheme with that of different C++ specific layouts. Our analytical cost model shows that in this data set, the new scheme is superior to the standard C++ layout and to the simple inlining algorithm [53]. Even though the new layout is not C++ specific, it competes favorably in this respect with aggressive inlining [53], arguably the best C++ layout scheme.

To better understand the intricacies of object layout, consider Figure 6.1a, which depicts a small single inheritance hierarchy.

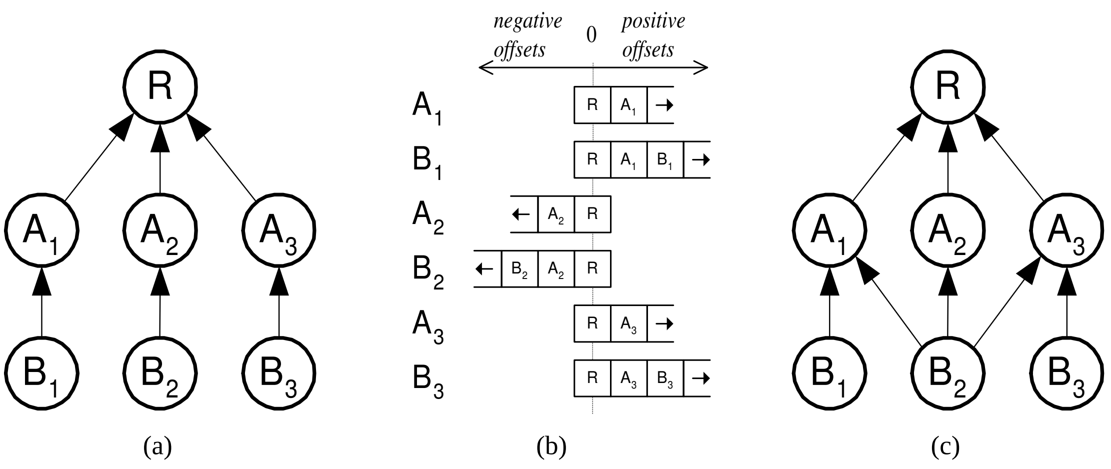


Figure 6.1: A small single inheritance hierarchy (a), a possible object layout for this hierarchy (b), and a multiple inheritance hierarchy in which there is no contiguous layout for all objects (c)

A possible object layout of the types defined in this hierarchy is shown in Figure 6.1b. The fields of A_1 are laid out just after R. The layout of B_1 adds its own fields in increasing offsets. All types inheriting from A_1 and B_1 will have positive directionality. Types A_2 and B_2 are laid out in negative offsets. This should also be the directionality of any of their descendants. Types A_3 and B_3 and all of their descendants have positive directionality.

Figure 6.1b demonstrates a degenerate case of the two-dimensional bi-directional layout scheme, in which there is only one layer. This layer is populated either in negative or positive offsets. In the general case, there are multiples layers, which may use for the same object type both positive and negative offsets, or even be empty.

Consider now the multiple inheritance hierarchy of Figure 6.1, obtained by adding multiple inheritance edges from B_2 to A_1 and A_3. Here and henceforth, inheritance is

[1]In the field dispatching technique we encapsulate fields in accessor methods.

assumed to be *shared* (**`virtual`** in the C++ jargon). Thus, in the figure, type B_2 has a *single* R sub-object. We believe that repeated inheritance, i.e., where type B_2 has two R sub-objects, is a rarity, or as one wrote: "repeated inheritance is an abomination".[2]

With the addition of multiple inheritance, a layout for B_2 becomes difficult, since at the same positive offsets immediately following R we expect to find both the fields of A_1 and the fields of A_3. This difficulty is no coincidence, and is in fact a result of the strong conformance requirement (or fixed offsets [113]) which we implicitly made:

> **The strong conformance requirement:** Every type must be laid out in the same offset in all of its descendants.

If the layout of A_1, A_2 and A_3 is required to be contiguous, then the fields of each of these types must be laid out adjacent to R. Since the layout of R in memory has only two sides, then it must be that at least two of A_1, A_2 and A_3 are laid out at the same side of R. This is not a problem as long as these two types are never laid out together, as is the case in single inheritance. The difficulty is raised in multiple inheritance, specifically, when there is a common descendant of these two types.

Thus, we see that it is sometimes impossible to maintain the strong conformance requirement and contiguous object layout. Our new scheme resolves the conflict by sacrificing the strong conformance requirement. In particular, each object is laid out in one or more layers, where each layer uses a bidirectional layout. The above difficulty is removed by placing (say) type A_3 in a different layer.

We note that separate compilation discovers too late that two base types compete for the same memory location, i.e., after the layout of these base types was determined. For this reason, our layout scheme, just as all other optimizing layouts, relies on whole program analysis.

Outline Pertinent definitions are given in Section 6.1, which also lists some of the standard simplifications of the object layout problem. Section 6.2 describes the criteria used in evaluating object layout schemes, using these to place our result in the context of previous work. The actual layout, which comes in three versions, is described in Section 6.3. Section 6.4 presents the algorithm for computing the actual layout. Section 6.5 describes the data-set used in the benchmark, while Section 6.6 gives the experimental results. Finally, conclusions and directions for future research are given in Section 6.7.

6.1 Definitions

Leading to a more exact specification of the problem, this section makes precise notions such as a hierarchy, incomparable types, and introduced and accessible fields in a type.

A *hierarchy* is specified by a set of types $\mathcal{T}$, $n = |\mathcal{T}|$, and a partial order, $\preceq$, called the *subtype relation* which must be reflexive, transitive and anti-symmetric. Let $a, b \in \mathcal{T}$ be arbitrary. Then, if $a \preceq b$ holds we say that a is a *subtype* of b and that b is a *supertype*

[2]words of an anonymous reviewer to [68]

of a. If neither $a \preceq b$ nor $a \succeq b$ holds, we say that the types are *incomparable*. Also, if there does not exist c such that $a \preceq c \preceq b$ and $c \neq a$, $c \neq b$, then we say that a is a *child* of b and that b is a *parent* of a.

A hierarchy is *single inheritance* if each $a \in \mathcal{T}$ has at most one parent, and *multiple inheritance* otherwise.

The set of ancestors of a type $a \in \mathcal{T}$ is $\text{ancestors}(a) \equiv \{b \in \mathcal{T} \mid a \preceq b\}$. We denote the number of ancestors of a by θ_a. Note that $a \in \text{ancestors}(a)$.

Types in a hierarchy may *introduce* fields, which can be thought of as unique names or selectors. We assume that there is no *field overriding*, i.e., that the same field name can only be used once in each type. Although C++ (and other languages) allow a derived class to reuse the name of a **`private`** field defined in a base class, our assumption is trivially satisfied by simple renaming.

Stated differently, our demand is that no run time dispatching process is required to select the particular “implementation” of a field name. This is precisely the case in statically typed languages, where the field name and the static object type uniquely determine the introducing class.

The problem of object layout in dynamically typed languages is not very interesting and excluded from the domain of discourse. In languages such as SMALLTALK, fields access is restricted to the methods of the defining object. With this restriction, the strong conformance requirement does not need to be satisfied[3]: The object layout problem then becomes trivial, even with the face of multiple inheritance. If however a dynamically typed language supports non-private fields, then there must be a runtime check that the accessed field is defined in the object. Such checks are related to subtyping tests and even to a more general dispatching problem which received extensive coverage in the literature [136–138].

For simplicity, we assume that all fields are of the same size. For a type $t \in \mathcal{T}$, let $|t|$ denote the number of fields introduced in t. The *accessible* fields of a type include all fields introduced in it and in any of its proper supertypes.

Given a type hierarchy, the *object layout problem* is to design a *layout scheme* for the objects of each of the types in the hierarchy, and a method for accessing at runtime the accessible fields of each type. Specifically, given a field `f` and an object address `o` of type t, the runtime system should be able to compute the address of `o.f`. The selector `f` is a compile time constant, while `o` is supplied only at runtime.

6.2 Previous Work

A layout scheme is evaluated by the following criteria.

1. *Dynamic memory overhead.* This is extra memory allocated for objects, i.e., memory beyond what is required for representing the object’s own fields. Ideally, this overhead is zero. However, holes in a noncontiguous object layout contribute to this

[3]In fact, even the weak conformance requirement (defined later in Section 6.2) is not satisfied.

overhead. Another overhead of this kind are compiler generated fields, e.g., virtual function table pointers (VPTRs) in C++.

Note that the semantics of most object oriented languages dictates that the layout of each object must include at least *one type identifier*. This identifier is used at runtime to identify the object type, for purposes such as dynamic message dispatch and subtyping tests. This identifier can be conveniently thought of as a field defined in a common root type (e.g., type `R` in Figure 6.1), and therefore is not counted as part of the dynamic memory overhead. However, if a scheme allocates multiple type identifiers, as is the case with the C++ standard layout, then all but the first identifier contribute to this overhead.

2. *Field access efficiency.* This is the time required to realize the field access operation `o.f`. Ideally, fields can be accessed in a single machine instruction, which relies on a fixed offset (from the object base) addressing mode. Layout schemes often rely on several levels of indirection for computing a field location in memory.

   It is common that all fields introduced in a certain type are laid out consecutively. Since `f` is supplied at compile time, the type t' in which `f` was introduced can be precomputed. The main duty of the runtime system is to find the location in memory in which the fields of t' are laid out in t, the type of `o`.

3. *Static memory overhead.* These are the tables and other data-structures used by the layout which are shared between all objects of a certain type. This overhead is usually less significant than the dynamic memory overhead, and therefore it seems worthwhile to maximize sharing. On the other hand, retrieving the shared information comes at the cost of extra indirections, and may reduce field access efficiency.

4. *Time for computing the layout.* This is the time required for computing the layout, which could be exponential in some schemes.

Object layout in a single inheritance hierarchy can simultaneously optimize all the above metrics. As can be seen in Figure 6.1b, both static and dynamic memory overheads are zero. Field access efficiency is optimal with no dereferencing. Also, the computation of the layout is as straightforward as it can be.

A trivial layout scheme for multiple inheritance which maintains the strong conformance requirement is that the layout of each type reserves memory for *all* fields defined in the hierarchy. Static memory overhead, time for computing the layout, and field access efficiency are optimized. However, dynamic memory overhead is huge since each object uses memory of size $\sum_{t \in \mathcal{T}} |t|$, regardless of its actual type, which usually has far fewer accessible fields.

Pugh and Weddell [113] investigated more efficient layout schemes which still fulfill the strong conformance requirement. The dynamic memory overhead of their main bidirectional object layout scheme is in one case study only 6%, compared to 47% in a unidirectional object layout. The authors also showed that the problem of determining whether an optimal bidirectional layout exists is NP-complete.

At the other extreme stands what may be called *field dispatching* layout scheme, which is employed by many dynamically typed programming languages including Cecil [26] and Dylan [120]. In this scheme, the layout of type t is obtained by iterating (in some arbitrary order) over the set $\mathrm{ancestors}(t)$, laying out their fields in order. Since the strong conformance property is broken, we encapsulate fields in accessor methods. If a field position changes in a subtype, we override its accessor. The dynamic memory overhead in this scheme is zero.

Dispatching on accessor methods can be implemented by an $n \times n$ *field dispatch matrix* which gives the base offset of a type in the layout of any of its descendants. This static memory overhead can be reduced if the matrix is compressed by e.g., techniques used for method dispatching (see e.g., [137] for a recent survey). A different implementation is found in the SmallEiffel compiler [135], in which a static branch code over the dynamic type of the object finds the required base offset.

The main drawback of field dispatching is in reduced field access efficiency. In the matrix implementation, field access requires at least three indirections in the simplest version, and potentially more with a compressed representation of the matrix.

An interesting tradeoff between the two extremes is offered by the memory model of C++ [93]. C++ distinguishes between **`virtual`** and *non-**`virtual`*** bases.[4] For non-virtual bases, C++ uses a relaxed conformance requirement. Let $t_1, t_2, t_3 \in \mathcal{T}$ be such that t_1 is a non-**`virtual`** base of t_2, and t_3 is an arbitrary subtype of t_2.

> **The weak conformance requirement:** The offset of t_1 with respect to t_2 is fixed in all occurrences of t_2 within $t_3 \preceq t_2$.

In other words, although the offset of t_1 is not the same in all of its descendants, it is fixed with respect to any specific descendant t_2, regardless of where that descendant is found. Consequently, to find the location of t_1 within t_3 it is sufficient to find the address of t_2 within t_3.

The weak conformance requirement can be maintained together with object contiguity in many multiple inheritance hierarchies, specifically those with no virtual-bases. However, since a type is not always located at the same offset, it is necessary to apply a process called **`this`**-adjustment [123] in order to access a field introduced in a supertype. For example, a method of t_2 cannot be invoked on an object of type t_3, without first correcting the pointer to the object, coercing it to type t_2.

The **`this`**-adjustment model incurs many penalties other than the time required for the addition. For example, the runtime system must apply null checks before a pointer can be corrected. Also, a conversion from an array of subtypes to an array of supertypes cannot be done constant time. Moreover, an object may contain multiple type-identifiers, (VPTRs in the C++ jargon) contributing to dynamic memory overhead. Also, the pointers to the same object may have different values which is a serious hurdle for garbage collectors (and for efficient identity testing).

In hierarchies with virtual bases, even the weak conformance requirement cannot be satisfied together with object contiguity. In these cases, C++ uses *virtual base pointers* (VBPTRs) to tie memory segments of the same object. Gil and Sweeney [68] give a

[4]We are not so interested in the textbook [124] difference between the two. Instead, we say that a type is a virtual base if two or more of its children have a common descendant.

detailed description of VBPTRs. We only mention that VBPTR can be stored directly in the objects, as in the "standard" C++ implementation, contributing to dynamic memory overhead, or moved to the static memory, at the cost of increasing field access time. Also, in order to be able to access fields at constant time, an implementation must store (a potentially quadratic number of) *inessential* VBPTRs. We note that referencing fields through VBPTRs also requires **`this`**-adjustment, and that a virtual base does have a VPTR.

Gil and Sweeney [68] proposed several optimizations of the standard C++ layout, which were then empirically evaluated by Eckel and Gil [53], whose main yardstick was dynamic and static memory overhead. The main optimization which contributes to field access efficiency is *simple-inline* which tries to reduce the number of virtual bases by conforming transformations of the hierarchy. *Aggressive-inline* does the same, using a maximal-independent set heuristic as procedure for finding a close to optimal set of transformations. The *bidirectional object layout* optimization reduces dynamic memory overhead but does not contribute to field access efficiency.

For the purpose of illustration, Figure 6.2 depicts a type hierarchy and its aggressive-inline C++ layout. The same hierarchy will be used below in Section 6.4 for demonstrating the new two-dimensional bi-directional layout. A C++ programmer is allowed to denote some of the inheritance edges as **`virtual`**. In the figure, inheritance edges $\langle \mathsf{B}, \mathsf{A} \rangle$ and $\langle \mathsf{C}, \mathsf{A} \rangle$ are virtual so that F has a single A sub-object. The virtual edges that were *inlined* in the aggressive-inline layout are marked in bold, while the other non-inlined virtual edges are dashed. The cells with a dot in Figure 6.2b represent VPTRs (VBPTRs were not drawn since they can be either shared in a class or duplicated in all of its instances).

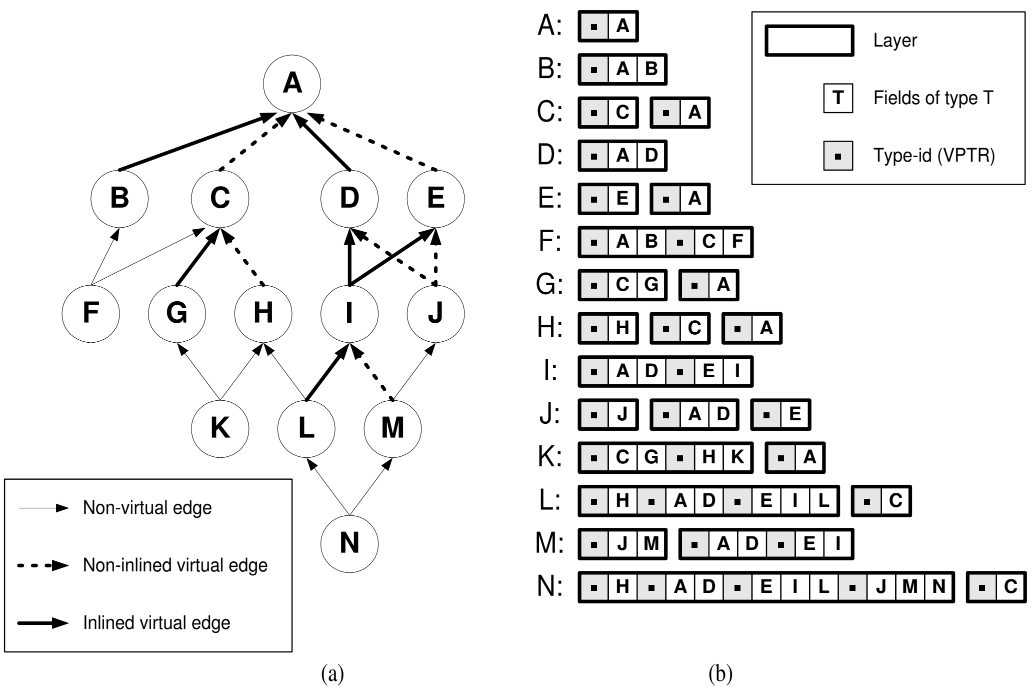


Figure 6.2: A type hierarchy (a) with its aggressive-inline C++ layout (b)

Our two-dimensional bi-directional scheme incurs *no dynamic memory overhead*. In this respect it is at least as good as any other layout scheme, and strictly better than all C++ implementations (which may include more than one VPTR). The most interesting criterion for comparison with C++ and field dispatching is therefore field access efficiency. We shall see that the our new scheme competes favorably even with the highly optimized and language specific aggressive-inline layout scheme.

Our results indicate that the time for computing the new layout is small—about 10 μSec per type (see Section 6.6). We also find that the static memory overhead is small compared both to field dispatching and various C++ techniques.

The new layout is *uniform*, in the sense that (unlike C++) the runtime system does not need any information on the static type of an object pointer in order to access any of its fields. Consider an object `o` and a field `f`. Then, the sequence of machine instructions for the field access operation `o.f` depends only on the selector `f`, and is the same regardless of the type of `o`. This is in contrast to languages such as C++ in which, depending on the static type of `o`, access to field `f` is either direct, or through indirection.

6.3 Two-Dimensional Bi-Directional Object Layout

In our two-dimensional bi-directional scheme, each field defined in the type hierarchy has a two-dimensional address $\langle \ell, \Delta \rangle$. Coordinate ℓ, $1 \leq \ell \leq L$, is the field's *layer*, where L is the number of layers used by the type hierarchy. (The assignment of types into layers is the subject of Section 6.4.) Coordinate Δ is an integral offset of the field in its layer. We say that the layout is bidirectional since this offset may be either positive or negative.

All fields introduced in the same type t are laid out consecutively: Their layer is the same as ℓ_t, the type's layer, while their offset is fixed with respect to Δ_t, the offset of the type. This section describes the actual object layout, which has three versions: the simple and not so efficient *canonical* layout, which is included for purpose of illustration, the general purpose *compact* layout, which we expect to be used in most cases, and the highly-optimized *inlined* layout which is applicable in some special cases.

In the *canonical* layout each object is represented as a pointer to a *Layers Dispatch Table* (LDT) of size L. Entry i, $i = 1, \ldots, L$, of the LDT points to the i^{th} layer of the object.

The canonical layout is demonstrated in Figure 6.3(a) for the case $L = 5$. The object depicted in the figure represented by a pointer p to its LDT, which stores pointers to layers L_1, L_3, and L_4. The type of the object is such that it has no fields from the second and the fifth layers. Hence the corresponding entries of the LDT are null.

In general, layers are two directional, and may store fields with both negative and positive offsets. Such is layer L_1 in the figure, with offsets in the range $-6, \ldots, +2$. However, the type of the object depicted has no fields with positive offsets in layer L_3. Similarly, layer L_4 has no fields with negative offsets.

We can see in the figure that each of the layers is contiguous. More precisely, if an object has a field at a certain layer in offset $\Delta > 0$, then it also has fields in all

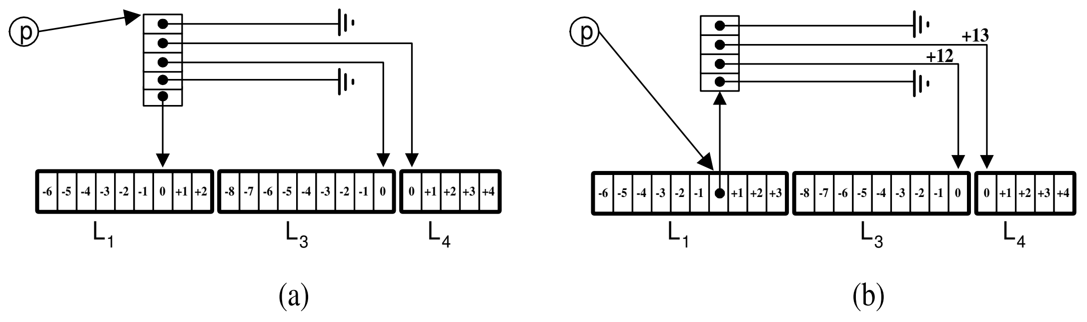


Figure 6.3: The canonical (a) and the compact (b) two-dimensional bi-directional layout of an object from a 5-layer hierarchy. Layers L_2 and L_5 are empty in the depicted object.

offsets $0, \ldots, \Delta - 1$. By placing the layers and the LDT next to each other we obtain a contiguous object layout. The pointers from the LDT to the layers can then be stored as relative offsets.

A compiler algorithm for producing the runtime access code in the canonical layout is presented in Algorithm 6.1. Take note that the type t, the layer ℓ_t, and the offsets Δ_t and $\Delta_{\texttt{f}}$ are computed at compile time. A *single* memory dereference is required to compute the field *address*.

Algorithm 6.1 An algorithm for generating field access code in the canonical layout

Given `f`, a name of a field of type **`int`**, and a pointer `p` to an object which uses the *canonical layout*, generate the code sequence (using pseudo-C++ notation) for accessing field `f` in `p`.

```
1: Let $t$ be the type in which f was defined
2: Let $\ell_t$ be the unique layer of $t$ // $\ell_t$ is a positive integer
3: Let integer $\Delta_t$ be the offset of $t$
4: Let $\Delta_{\texttt{f}}$ be the offset of f within its type // $\Delta_{\texttt{f}}$ is a non-negative integer
5: Output
       int *layer_ptr = ((int **)p)[$\ell_t - 1$];
       int &r = layer_ptr[$\Delta_t$ + $\Delta_{\texttt{f}}$];
```

It is important to notice that the occupied entries in each layer depend only on the object *type*. Therefore, an offset-based LDT is identical in all objects of the same type and can be shared. The *compact* version of object layout is obtained by employing this sharing and by letting the object pointers reference the first layer directly, which tends to be the largest in our algorithm for assigning fields to layers.

Figure 6.3b gives an example of the compact layout of the same object of Figure 6.3a. In the figure we see the same three non-empty layers: L_1, L_3 and L_4. However, the object pointer p now points to offset 0 in layer L_1. At this offset we find the *object type identifier*, which is a pointer to the shared LDT. Notice that the size of layer L_1 was increased by one to accommodate the object type identifier. Also, there are now only four entries in the LDT, which correspond to layers $L_2, \ldots, L_5$.

Algorithm 6.2 is run by the compiler to generate the code sequence for accessing a field in the compact layout. If the compiler determines that the field is in the first

Algorithm 6.2 An algorithm for generating field access code in the compact layout

Given `f`, a name of a field of type **`int`**, and a pointer `p` to an object which uses the *compact layout*, generate the code sequence (using pseudo-C++ notation) for accessing field `f` in `p`.

```
1: Let $t$ be the type in which f was defined
2: Let $\ell_t$ be the unique layer of $t$ // $\ell_t$ is a positive integer
3: Let integer $\Delta_t$ be the offset of $t$
4: Let $\Delta_f$ be the offset of f within its type // $\Delta_f$ is a non-negative integer
5: If $\ell_t = 1$ then
6:    Output
          int &r = ((int *)p)[$\Delta_t$ + $\Delta_f$];
7: else
8:    Output
          int *p1 = *((int **)p);
          int layer_offset = p1[$\ell_t - 2$];
          int &r = p[layer_offset + $\Delta_t$ + $\Delta_f$];
9: fi
```

layer, then the field can be accessed directly—*no* memory dereferences are required for computing its address. If the field however falls in any other layer, then memory must be dererenced once to find the LDT, and then again to find the layer offset. Also, in this case, the addressing mode for the final field access is slightly more complicated since it must add compile- and runtime- offsets.

The LDT in the example of Figure 6.3 includes only four entries, all of which are byte-size integers (assuming of course that the object size is less than 256 bytes). The entire LDT can be represented as a single 32 bit words. The *inlined* layout is obtained from the compact layout by inlining the LDT into the object's first layer. At the cost of increasing object space, inlining saves a level of indirection in fetching LDT entries. Note that even if the LDT is stored inside the object, each object must include at least one type identifier for purposes such as subtyping tests and dispatching. Therefore, even in this simple example, the inlined layout uses more space than the compact layout.

6.4 Computing Type Addresses

This section is dedicated to the algorithm for assigning field addresses. The main constraint to maintain is that all layers are contiguous in all types. It is always possible to find such an assignment, since each field can be allocated its own layer (as done in field dispatching).

Our objective is an assignment which minimizes L, the number of layers. One reason for doing so, is that the memory required for LDTs is $L \times n$. LDTs are source for static memory overhead in the compact layout, and dynamic memory overhead in the inlined layout.

However, our most important motivation is reducing the *likelihood of LDT fetches*, or in other words, inefficiency of field access. If the number of layers is one, then all fields can be retrieved without any dereferences. We note that if the number of layers is small, then an optimizing compiler might be able to pre-fetch and reuse layer addresses to accelerate field access.

Note first that each layer has a positive and a negative *semi-layer*, and that these semi-layers are independent for the purpose of allocation. To understand the constraints of allocation better, consider Figure 6.4a which gives the object layout for our running example.

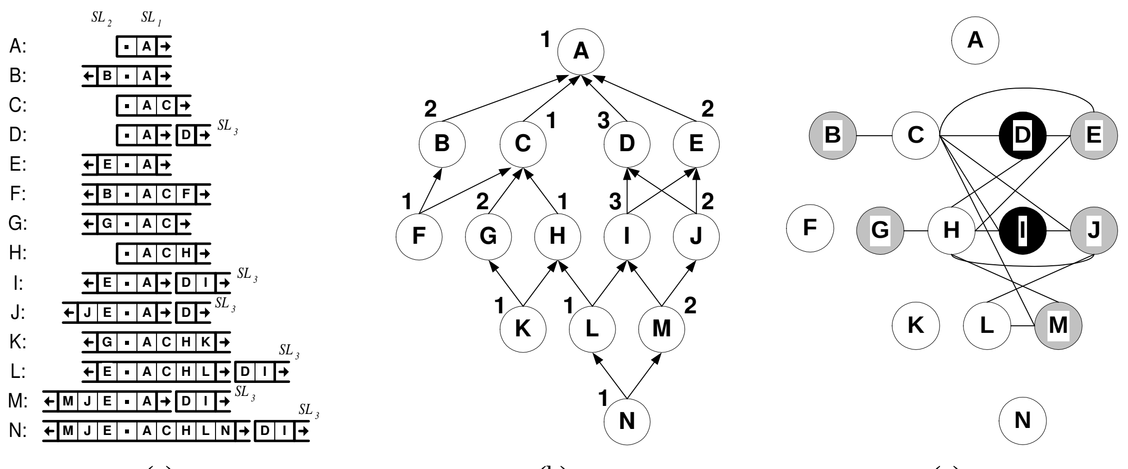


Figure 6.4: The two-dimensional bi-directional object layout of the running example (a), the allocation of types in it to semi-layers (b), and the conflict graph with its coloring (c)

We see in the figure that the hierarchy uses a total of two layers and three semi-layers. The first layer has at offset 0 the object type identifier and a positive and negative semi-layers. The second layer uses only the positive semi-layer. The arrows in the figure indicate the place where the semi-layer may continue.

Figure 6.4b shows the allocation of types to semi-layers which generates this layout: Seven types A, C, F, H, K, L, and N are in semi-layer 1 (positive side of the first layer). Semi-layer 2 (negative side of the first layer) includes five types: B, E, G, J, and M. Only D and I are in semi-layer 3 (positive side of the second layer). The layout of type N for example, makes use of all three semi-layers, while the layout of D uses just semi-layers 1 and 3.

Notice the following points: **(i)** Semi-layers 1 and 2 comprising the first layer are in a fixed offset. Semi-layer 3 occurs at different offsets in different types. **(ii)** Each type is always placed in the same location in its layer. For example, E is located in the first location in semi-layer 2 in the layouts of all of its descendants: E, I, J, L, M, and N. **(iii)** The same location in the same semi-layer can be used for different types. For example, the first location of semi-layer 1 stores also the fields of B in the layout of F, and the fields of G in the layout of K. **(iv)** Types are allocated to semi-layers in descending subtyping order. For example, we see that types A, C, H, L and N are placed in this order in semi-layer 1 in the layout of N and that $\mathsf{A} \succeq \mathsf{C} \succeq \mathsf{H} \succeq \mathsf{L} \succeq \mathsf{N}$.

The general question is whether two arbitrary types $a, b \in \mathcal{T}$ can be allocated to the same semi-layer, and what should their relative ordering in that semi-layer should be. Suppose first, without loss of generality, that $a \preceq b$. Then, whenever a appears, so does b. Therefore, with the absence of other constraints, we can allocate a and b into the same semi-layer, and a must be placed after b in this semi-layer. If however a and b are incomparable, then they could be allocated to the same semi-layer, and even to the same location in the level, as long as they do not occur together in the layout of any third type c. In other words, the allocation is allowed as long as a and b have no common descendants.

Figure 6.4c shows the *conflict graph* of our running example, where two types are connected by an edge if they are incomparable, yet have a common descendant. We see in the figure that no edges are incident on A. This is because A is the root, and as such is comparable with all types in the hierarchy. Also, no edges are incident on the leaves F, K and N. The edge between C and E , for example, is due to their common descendant L.

A node coloring of the conflict graph provides a legal allocation. We of course seek a minimal coloring of this graph. Figure 6.4c gives a coloring of the conflict graph of the running example. A total of three colors are used: White nodes are allocated to semi-layer 1, grey to semi-layer 2, and black to semi-layer 3.

Algorithm 6.3 shows the general procedure for address allocation. Using a graph coloring heuristic, the algorithm computes the number of layers for the layout. Also, for each type t in the input hierarchy the algorithm returns ℓ_t, its layer and Δ_t, the base offset in the layer at which its fields are allocated. If $\Delta_t \geq 0$, then fields are allocated in ascending addresses. Otherwise, t is in the negative semi-layer, and field are placed in the addresses below Δ_t.

Lines 1–10 compute the edges in the conflict graph. In the main loop, we consider the ancestors of each candidate. There is a conflict between any two of its ancestors if they are incomparable. The runtime of the inner loop should (empirically) be close to linear, since the average number of ancestors in our hierarchies is small.

Next (lines 11–12) we compute the conflict graph and a coloring of it. We use a simple, greedy heuristic for finding this coloring. (We color nodes with larger degree first, using the first available color.) A favorable property of this heuristic is that the color groups tend to come out in descending order, i.e., $|\phi^{-1}(i)| \geq |\phi^{-1}(i+1)|$ for $i = 1, \ldots, s-1$. Since fields in the first layer can be accessed in a single indirection, the first layer should be as large as possible.

The next command block computes the layer of each type t, and its (positive or negative) offset within this layer. Lines 14–19 compute the total size of types which precede t in its semi-layer. After computing the layer number (line 20) we turn to making the necessary corrections to the offset. In general, positive semi-layers use offsets $0, +1, +2, \ldots$, while negative semi-layers use offsets $-1, -2, \ldots$ (lines 21–22). However, layer 1 is special since it contains the type identifier at offset 0 (lines 23–24).

6.5 Data Set

For the purpose of evaluating the multi-layer object layout scheme, we used an ensemble of 28 type hierarchies, drawn from eight different programming languages, and spanning

Algorithm 6.3 Produce the compact two-dimensional bi-directional layout of a hierarchy

Given a hierarchy $\mathcal{T}$ and $\preceq$, return the number of layers L, and compute ℓ_t and Δ_t for each type $t \in \mathcal{T}$

```
1: Let $E \leftarrow \emptyset$ // $E$ is the set of edges in the undirected conflict graph
2: For all $t \in \mathcal{T}$ do // Consider all possible common descendants
3:    For all $p_1, p_2 \in \mathrm{ancestors}(t)$ do // $p_1$ and $p_2$ have a common descendant $t$
4:       If $p_1 \not\preceq p_2$ and $p_1 \not\succeq p_2$ then // $p_1$ and $p_2$ are incomparable
5:          If $\{p_1, p_2\} \notin E$ then // A new conflict edge found
6:             $E \leftarrow E \cup \{\{p_1, p_2\}\}$
7:          fi
8:       fi
9:    od
10: od

11: Let $G \leftarrow \langle \mathcal{T}, E \rangle$ // $G$ is the graph of conflicts between types
12: Let $\phi : \mathcal{T} \mapsto [1, \ldots, s]$ be a coloring of the nodes of $G$

13: For all $t \in \mathcal{T}$ do // Compute the offset and the layer of $t$
14:    $\Delta_t \leftarrow 0$ // Compute the total size of proper ancestors in the same semi-layer as $t$
15:    For all $p \in \mathrm{ancestors}(t), p \neq t$ do
16:       If $\phi(p) = \phi(t)$ then // Ancestor $p$ is in the same semi-layer as $t$
17:          $\Delta_t \leftarrow \Delta_t + |p|$
18:       fi
19:    od
20:    $\ell_t \leftarrow \lceil \phi(t)/2 \rceil$ // Layer $l$ hosts colors $2l - 1$ and $2l$
21:    If $\phi(t) \bmod 2 = 0$ then // Even colored objects are laid out in negative semi-layers
22:       $\Delta_t \leftarrow -\Delta_t - 1$ // Offsets of negative semi-layers start at $-1$
23:    else if $\phi(t) = 1$ then
24:       $\Delta_t \leftarrow \Delta_t + 1$ // Offset 0 in layer 1 is reserved for the type-identifier
25:    fi
26: od

27: Return $\lceil s/2 \rceil$
```

almost 50,000 types. The first 27 hierarchies[5] were used in our previous benchmarks. A detailed description of their origin, respective programming language, and many of their statistical and topological properties can be found elsewhere [136, 137]. (Even though multiple inheritance of fields is not possible in JAVA, the JAVA hierarchies are still useful in characterizing how programmers tend to use multiple inheritance.) To these we added Flavors, a 67-type hierarchy representing the *multi-inheritance core* of the Flavors language [98] benchmark used by Pugh and Weddell [113, Fig. 5].

Together, the hierarchies span a range of sizes, from 67 types (in IDL and Flavors) up to 8,793 types in MI: IBM SF, the median being 930 types. The hierarchies are relatively

[5]IDL, MI: IBM XML, JDK 1.1, Laure, Ed, LOV, Cecil2, Cecil-, Unidraw, Harlequin, MI: Orbacus Test, MI: HotJava, Dylan, Cecil, Geode, MI: Orbacus, Vor3, MI: Corba, JDK 1.18, Self, Vortex3, Eiffel4, MI: Orbix, JDK 1.22, JDK 1.30, MI: JDK 1.3.1, and MI: IBM SF.

shallow, with heights between 9 and 17. Most types have just one parent, and the overall average number of parents is 1.2. In these and other respects, the hierarchies are not very different from balanced binary trees [53].

The number of ancestors is typically small, averaging less than 10 in most hierarchies. Exceptions are the Geode and the Self hierarchies, which make an extensive use of multiple inheritance. In Geode, there are 14 ancestors in average to each type, and there exists a type with as many as 50 ancestors. Self has 31 ancestors in average per type. The topology of Self is quite unique in that almost all types in it inherit from a type with 23 ancestors. Table 6.1 below gives (among other information), the number of types in each hierarchy, and the maximal and average number of ancestors.

6.6 Experimental Results

This section presents the results of running Algorithm 6.3 on our data set. Since this algorithm depends on a graph-coloring heuristic (Line 12), we would like first to be assured by the output quality. We remind the reader that if a graph has a clique of size k, then it cannot be colored by fewer than k colors. Although it is not easy to find cliques in general graphs, some cliques can be efficiently found in conflict graphs. Consider a type t and its set of ancestors $\text{ancestors}(t)$. Let $P_t \subseteq \text{ancestors}(t)$ be a set of types which are pair-wise incomparable. Then any $t_1, t_2 \in P_t$ are in conflict, and the set P_t is a clique in the conflict graph. Finding a maximal set of incomparable nodes in a hierarchy is a standard procedure of finding a maximal anti-chain in a partial order [126].

Table 6.1 compares the number of colors and layers with the predictions of the lower bound thus found.

Let $\omega_t = \max\{|P_t| \mid P_t \subseteq \text{ancestors}(t)$ is a set of pair-wise incomparable types$\}$, i.e., ω_t is the size of the maximal anti-chain among the ancestors of t. Then, $\omega = \max_{t \in \mathcal{T}}\{\omega_t\}$ is a lower bound on the number of colors (or semi-layers), and $\lceil \omega/2 \rceil$ is a lower bound on the number of layers L. We see in the table that $s > \omega$ only in seven hierarchies: Flavors, Ed, LOV, MI: Orbacus Test, MI: HotJava, Geode and MI: Corba. In these seven cases, $s = \omega + 1$, so the number of colors was off by at most one from the lower bound. Further, as the next two columns indicate, the situation that the number of layers is greater than the prediction of the lower bound, occurs in only three hierarchies: Ed, MI: HotJava and MI: Corba.

It is also interesting to compare the number of colors and the number of layers with the maximal number of ancestors, denoted $\alpha = \max(\theta_t)$. As expected, the number of colors is never greater than the maximal number of ancestors, and is typically much smaller than it. The number of entries in the LDT is even smaller, since every two colors are mapped to a single layer.

The maximal number of layers in the field dispatching technique is exactly α, since each layer is a singleton. The field dispatch matrix can be compressed using method dispatching techniques, such as selector coloring [44, 118]. A lower bound on the space requirement of selector coloring is $n \times \alpha$. We therefore have that the static memory of our layout scheme $n \times L$ is superior to that of the field dispatch matrix compressed using

Hierarchy $\langle \mathcal{T}, \preceq \rangle$	$n = \lvert\mathcal{T}\rvert$	ω [a]	s [b]	$\lceil \omega/2 \rceil$	$\lceil s/2 \rceil$	$\max(\theta_t)$[c]	$\mathrm{avg}(L_t)$[d]	$\mathrm{avg}(\theta_t)$[e]
Flavors	67	3	4	2	2	13	1.6	4.9
IDL	67	2	2	1	1	9	1.0	4.8
MI: IBM XML	145	5	5	3	3	14	1.5	4.4
JDK 1.1	226	2	2	1	1	8	1.0	4.2
Laure	295	3	3	2	2	16	1.1	8.1
Ed	434	12	13	6	7	23	3.2	8.0
LOV	436	13	14	7	7	24	3.5	8.5
Cecil2	472	8	8	4	4	29	2.0	7.4
Cecil-	473	8	8	4	4	29	2.0	7.4
Unidraw	614	3	3	2	2	10	1.0	4.0
Harlequin	666	14	14	7	7	31	1.9	6.7
MI: Orbacus Test	689	3	4	2	2	12	1.3	3.9
MI: HotJava	736	14	15	7	8	23	2.0	5.1
Dylan	925	3	3	2	2	13	1.1	5.5
Cecil	932	6	6	3	3	23	1.7	6.5
Geode	1,318	21	22	11	11	50	5.1	14.0
MI: Orbacus	1,379	11	11	6	6	19	1.6	4.5
Vor3	1,660	6	6	3	3	27	1.6	7.5
MI: Corba	1,699	6	7	3	4	18	1.3	3.9
JDK 1.18	1,704	12	12	6	6	16	1.2	4.3
Self	1,802	24	24	12	12	41	10.7	30.9
Vortex3	1,954	8	8	4	4	30	1.7	7.2
Eiffel4	1,999	15	15	8	8	39	2.2	8.8
MI: Orbix	2,716	6	6	3	3	13	1.1	2.8
JDK 1.22	4,339	14	14	7	7	17	1.5	4.4
JDK 1.30	5,438	15	15	8	8	19	1.5	4.4
MI: JDK 1.3.1	7,401	21	21	11	11	24	1.5	4.4
MI: IBM SF	8,793	13	13	7	7	30	2.3	9.2

[a]the maximal size of an anti-chain in the ancestors of any type $t \in \mathcal{T}$

[b]the number of colors (or semi-layers) used by Algorithm 6.3

[c]$\max\{\theta_t \mid t \in \mathcal{T}\}$

[d]$\frac{1}{n}\sum_{t \in \mathcal{T}} L_t$

[e]$\frac{1}{n}\sum_{t \in \mathcal{T}} \theta_t$

Table 6.1: Statistics on the input hierarchies, including the number of colors and layers found by Algorithm 6.3 compared with the maximal anti-chain lower bound

selector coloring.

The next two columns of Table 6.1 give another comparison of hash-table implementation of the LDT with a hash table implementation of the field dispatch matrix. We see that the number of layers which each object uses is typically small. No more than 3.5 in all but the Self and Geode hierarchies. In all hierarchies, we see that the average number of ancestors is much greater than the average number layers. This shows that **(i)** Algorithm 6.3 is successful in compressing multiple types into layers, and consequently that **(ii)** the LDT places weaker demands than the field dispatch matrix on static memory.

The theoretical complexity of Algorithm 6.3 is $O(n^3)$, since lines 2–3 may iterate in certain hierarchies over a fixed fraction of all possible type triplets. The runtime of the simple greedy graph-coloring heuristic is $O(n^2)$. In practice however, the algorithm runs much faster. By applying some rather straightforward algorithmic optimizations, e.g., considering in line 2 only types which have more than one parent, the run times were reduced even further.

On a Pentium III, 900Mhz machine, equipped with 256MB internal memory and running a Windows 2000 operating system, Algorithm 6.3 required less than 10 mSec in 19 hierarchies. Seven hierarchies required between 10 mSec and 50 mSec. The worst hierarchy was MI: IBM SF which took 400 mSec. The total runtime for all hierarchies was 650 mSec, which gives on average 13μSec of CPU time per type. The runtime of C++ aggressive-inline procedure on the same hardware is much slower. For example, aggressive inline of MI: IBM SF took 3,586 mSec, i.e., about 9 times slower. Simple inline of MI: IBM SF took 2,294 mSec, which is still much slower.

The most important criterion for evaluating a layout scheme is field access efficiency.

Since the hierarchies were drawn from different languages and were not associated with any application programs, we were unable to directly measure the actual cost of field access in the various layout schemes. We can however derive other metrics to compare the costs of the new layout technique with that of prior art.

For example, the number of layers used by a given type, gives an indication on the number of different dereferences required to access *all* the object fields. The corresponding metric in C++ is the number of virtual bases, which can be accessed only by dereferencing a VBPTR.

Figure 6.5 compares the average number of layers of the new scheme with that of the standard C++ implementation, the simple inlined implementation and the aggressive inlined implementation. In making the comparison we bear in mind that the new scheme is both language-independent and space-optimal—properties which the C++ schemes do not enjoy.

We see in the figure that with the exception of Self hierarchy (which as we mentioned above has a very unique topology), the new layout scheme is always superior to the standard- and simple-inlined implementation of C++. Moreover, the new scheme is superior or comparable with the aggressive-inline layout scheme, with the exception of four hierarchies: Ed, LOV, Geode and Self. Comparing the *maximal-* rather than the *average-* number of layers yields similar results.

Table 6.2 shows the extra dynamic memory consumed by the various C++ layout schemes, specifically for VPTRs.

Curiously, the four hierarchies in which the new scheme does not perform as well, Ed, LOV, Geode and Self, are exactly the hierarchies in which the C++ schemes, including the highly optimized aggressive inline waste the most amount of dynamic memory.

We also offer a more sophisticated theoretical model for comparing the performance of various schemes of object layout which involve indirection to access various fields. Suppose that a certain field was retrieved from a certain layer. Then, a good optimizing compiler should be able to reuse the address of this layer in retrieving other fields from this layer. Even in the standard C++ layout, the compiler may be able to reuse the address

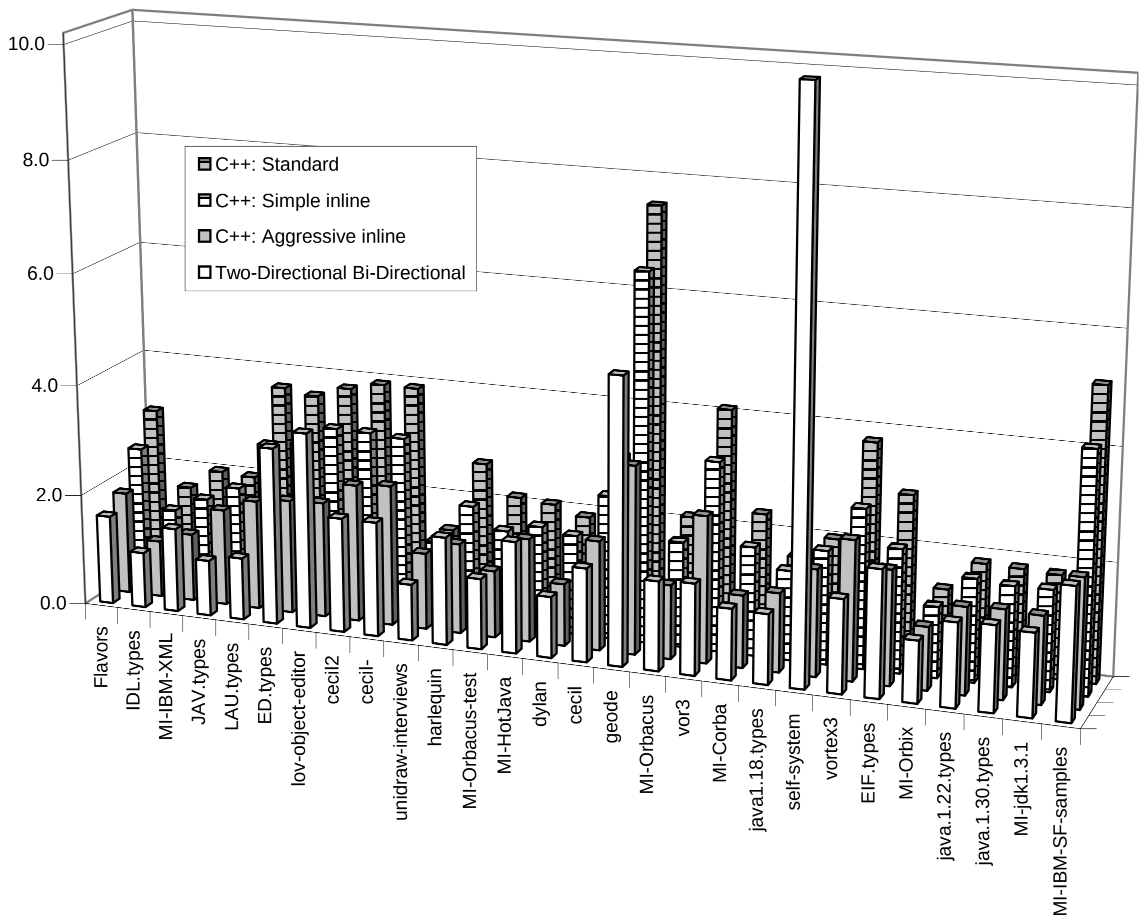


Figure 6.5: Average no. of layers in different hierarchies

of a virtual base to fetch additional fields from this base.

For a fixed type t, and for a sequence of k field accesses, we would like to compute $A_t(k)$, the expected number of extra dereferences required to access these fields. Since much empirical data is missing from our ensemble of hierarchies, we were inclined to make two major simplifying assumptions:

1. *Uniform class size.* The number of fields introduced in each type is the same. Although evidently inaccurate, this assumption should not be crucial to the results. We do expect that most classes introduce a small number of fields, with a relatively small variety.

2. *Uniform access probability.* The probability of accessing any certain field is fixed, and is independent of the fields accessed previously, nor of the type in which the field is defined. This assumption is clearly in contradiction to the *principle of locality of reference.*

   However, as we shall see, locality of reference improves the performance of layout schemes. It is not clear whether this improvement contribute more to any specific scheme.

The θ_t ancestors of t are laid out in L_t different layers or virtual bases, such that layer i

Hierarchy	Average			Median			Maximum		
	C++	S-In	A-In	C++	S-In	A-In	C++	S-In	A-In
Flavors	3.4	3.2	2.4	3	3	2	9	8	5
IDL	1.9	1.6	1.2	2	2	1	3	2	2
MI: IBM XML	2.8	2.8	2.0	2	2	1	9	9	6
JDK 1.1	2.1	2.0	1.8	2	2	2	4	4	3
Laure	3.9	3.2	2.3	4	3	2	8	7	5
Ed	5.2	5.0	4.2	4	4	4	16	16	12
LOV	5.6	5.5	4.6	5	5	4	17	17	13
Cecil2	4.6	4.4	3.4	3	3	3	17	15	9
Cecil-	4.6	4.3	3.5	3	3	3	17	15	9
Unidraw	1.4	1.4	1.4	1	1	1	4	3	3
Harlequin	3.6	3.2	2.7	2	2	2	21	19	16
MI: Orbacus Test	2.5	2.1	1.7	2	2	1	8	6	5
MI: HotJava	2.9	2.9	2.7	2	2	2	17	17	15
Dylan	2.0	1.9	1.3	2	2	1	7	6	5
Cecil	3.7	3.5	2.7	3	3	2	16	13	8
Geode	9.9	9.5	8.3	9	9	7	32	31	27
MI: Orbacus	2.8	2.6	2.2	2	2	1	13	12	11
Vor3	4.6	4.2	3.5	4	3	3	17	14	11
MI: Corba	2.6	2.3	1.7	2	2	1	14	12	10
JDK 1.18	1.9	1.9	1.7	2	2	1	14	13	12
Self	21.2	21.2	21.1	22	22	22	26	25	25
Vortex3	4.4	3.8	3.4	3	3	3	18	15	11
Eiffel4	3.7	3.4	3.1	2	2	2	20	17	16
MI: Orbix	1.5	1.4	1.3	1	1	1	7	7	6
JDK 1.22	2.4	2.3	2.1	2	2	2	16	15	14
JDK 1.30	2.4	2.3	2.1	2	2	2	17	17	16
MI: JDK 1.3.1	2.3	2.3	2.0	2	2	1	23	22	21
MI: IBM SF	5.8	5.8	3.6	6	6	3	16	16	13
Total	4.2	4.0	3.3	-	-	22	32	31	27
Median	3.2	3.0	2.4	2	2	2	16	14.5	11
Minimum	1.4	1.4	1.2	1	1	1	3	2	2
Maximum	21.2	21.2	21.1	22	22	22	32	31	27

Table 6.2: No. of VPTRs using standard C++ layout, simple inline (S-In), and aggressive inline (A-In)

(virtual base i) has $\theta_t(i)$ ancestors. The first layer can always be accessed directly. Access to a field in layer i in step k requires a dereference operation, if that layer was not accessed in steps $1, \ldots, k-1$.

Let $X_t(i)$, $i = 2, \ldots, L_t$ be the random binary variable which is 1 if a field of level i was not referenced in any of the steps $1, \ldots, k$. Then,

$$\textbf{Prob}[X_t(i) = 1] = \textbf{Exp}(X_t(i)) = \left(1 - \frac{\theta_t(i)}{\theta_t}\right)^k .$$

Additivity of expectation allows us to sum the above over i, obtaining that the expected number of levels (other than the first) which were not referenced is

$$\sum_{i=2}^{L_t} \left(1 - \frac{\theta_t(i)}{\theta_t}\right)^k .$$

Using the linearity of expectation, we find that the expected number of referenced levels, i.e., the number of dereferences is simply

$$A_t(k) = (L - 1) - \sum_{i=2}^{L_t} \left(1 - \frac{\theta_t(i)}{\theta_t}\right)^k . \tag{6.1}$$

Averaging over an entire type hierarchy, we define

$$A(k) = \frac{1}{n} \sum_{t \in \mathcal{T}} A_t(k) \tag{6.2}$$

Figure 6.6 gives a plot of $A(k)$ vs. k in four sample hierarchies in the layout schemes field dispatching, standard C++ layout, simple inline (S-Inline), aggressive inline (A-Inline), and our two-dimensional bi-directional layout (TDBD). Values of $A(k)$ were computed using (6.1) and (6.2) in the respective hierarchy and object layout scheme. For field dispatching, we set $\theta_t(i) = 1$.

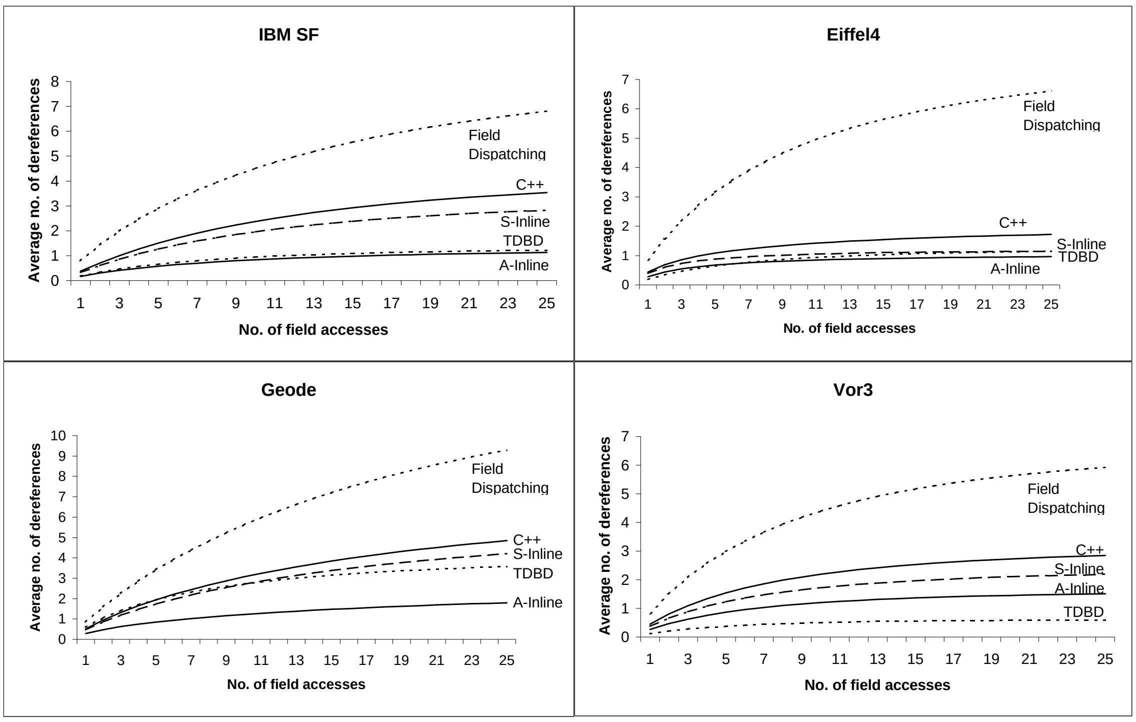


Figure 6.6: Average no. of dereferences vs. no. of field accesses in four hierarchies

It is interesting to see that in all hierarchies and in all layout schemes, the expected number of dereferences is much smaller than the number of actual fields accessed. It is also not surprising that $A(k)$ increases quickly at first and slowly later. As expected, the new scheme is much better than field dispatching. The graphs give hope of saving about 75% of the dereferences incurred in field dispatching. (Note however that the model does not take into account any optimizations which runtime systems may apply to field dispatching.)

The other, C++ specific techniques are also more efficient than field dispatching. We now turn to comparing these with our scheme. In the Vortex3 hierarchy the new scheme dramatically improves the expected number of dereferences compared to any of the C++ layout schemes. The new scheme is also the best in smaller k values in the Eiffel4 hierarchy, and is comparable to aggressive inline with greater values of k. Another typical behavior is demonstrated by MI: IBM SF, in which the new scheme is almost the same as aggressive-inline. In the Geode hierarchy which is one of the two hierarchies in which the two-dimensional bi-directional scheme cannot find a good partitioning into a small number of layers, we find that aggressive inline gives the best results in terms of field access efficiency. Still, even in this hierarchy the new scheme is better than the standard C++ implementation and the simple-inline outline heuristic.

6.7 Conclusions and Open Problems

The two-dimensional bi-directional object layout scheme enjoys the following properties: **(i)** the dynamic memory overhead per object is a single type-identifier, **(ii)** the static memory per type is small: at most 11 cells in our data set, but usually only around 5 cells, **(iii)** small time for computing the layout: an average of 13 μSec per type in our data set, and **(iv)** good field access efficiency as predicted by our analytical model: the new scheme always improves upon the field dispatching scheme and on the standard C++ layout model. Even compared to the highly optimized C++ layout, after performing aggressive inline, the new scheme still compares favorably.

We note that the new scheme does not rely on `this`-adjustment, and in the few hierarchies where the aggressive-inline of C++ won, it was with the cost of large dynamic memory overheads, e.g., as much as 21 VPTRs on average in the Self hierarchy.

The one-dimensional bi-directional layout of Pugh and Weddell's [113] realizes field access in a single indirection, but it may leave holes in some objects. In comparison, our two-dimensional bi-directional layout has no dynamic memory overheads, but a field access might require extra dereferences. In the Flavors hierarchy Pugh and Weddell reported 6% dynamic memory overhead (assuming a single instance per type). Our scheme uses only two layers for this hierarchy, and the probability that a field access would require extra dereferences is only 0.19.

Directions for future work include empirical study of frequencies of field accesses, and further reducing the static memory overheads. In dynamically typed languages where fields can be overloaded, the layout algorithm must color fields instead of types. Empirical data should be gathered to evaluate the efficiency of the layout algorithm in such languages.

Chapter 7

Efficient Algorithms for Isomorphisms of Simple Types

Chapter Summary
The first order isomorphism problem *is to decide whether two non-recursive types using product- and function-type constructors, are isomorphic under the axioms of commutative and associative products, and currying and distributivity of functions over products. We show that this problem can be solved in $O(n \log^2 n)$ time and $O(n)$ space, where n is the input size. This result improves upon the $O(n^2 \log n)$ time and $O(n^2)$ space bounds of the best previous algorithm. We also describe an $O(n)$ time algorithm for the* linear isomorphism problem*, which does not include the distributive axiom, thereby improving upon the $O(n \log n)$ time of the best previous algorithm for this problem.*

It is a matter of basic high school algebra to prove the equality

$$\left((\mathsf{ab})^{\left(\mathsf{a}^{\mathsf{b}}\right)}\right)^{\left(\mathsf{b}^{\mathsf{a}}\right)} = \mathsf{a}^{\mathsf{a}^{\mathsf{b}}\mathsf{b}^{\mathsf{a}}}\mathsf{b}^{\mathsf{b}^{\mathsf{a}}\mathsf{a}^{\mathsf{b}}}. \tag{7.1}$$

Yet, as we shall see in this chapter, a systematic and efficient production of such a proof is non-trivial. With the familiar perspective of viewing multiplication as product-types, exponentiation as function-types, and variables as primitive-types, (7.1) becomes an instance of a simple, i.e., non-recursive, type isomorphism problem. In its turn, type isomorphism has close connections to category theory [19, 122] and intuitionistic logic [79].

The isomorphism variant which concerns us here is characterized by commutativity and associativity of products, and currying and distributivity of functions over products. This variant has practical interest in the context of the search for compatible functions in function libraries.[1] (A detailed treatise of this application can be found in Di Cosmo's book [39], which discusses also extensions to second order types and the ML type theory.)

More formally, we consider the set of first order isomorphisms holding in all models of the lambda calculus with product-types (surjective pairing), function-types, and unit

[1]Besides being sufficient for the proof of equations such as (7.1).

types, as defined by the following *general grammar*

$$\tau ::= \mathbf{T} \quad | \quad x \quad | \quad \tau \to \tau \quad | \quad \tau \times \tau \quad ,$$

where $\mathbf{T}$ is the unit type, x stands for an arbitrary primitive-type, $\to$ denotes a function-type, and $\times$ denotes a product-type.

In defining the isomorphism relation we shall use the following seven axiom schemas.

$(\mathcal{A}.1)$	$A \times \mathbf{T} = A$	
$(\mathcal{A}.2)$	$A \to \mathbf{T} = \mathbf{T}$	
$(\mathcal{A}.3)$	$\mathbf{T} \to A = A$	
$(\mathcal{A}.4)$	$A \times B = B \times A$	(Commutative)
$(\mathcal{A}.5)$	$A \times (B \times C) = (A \times B) \times C$	(Associative)
$(\mathcal{A}.6)$	$(A \times B) \to C = A \to (B \to C)$	(Currying)
$(\mathcal{A}.7)$	$A \to (B \times C) = (A \to B) \times (A \to C)$	(Distributive)

(Here and henceforth, the range of variables A, B and C is any type expression in the general grammar.)

For a long time, the problem of deciding first order isomorphisms of simple types was thought to require exponential time [19]. It was recently shown [32] that the variant of our interest can be decided in $O(n^2 \log n)$ time and $O(n^2)$ space, where n is the length of some standard representation of the two input types. The main contribution described in this chapter is an improvement of this result to $O(n \log^2 n)$ time and $O(n)$ space. We also give algorithms using $O(n)$ time and space for important special cases.

The arithmetic version of these seven axioms (substituting multiplication, exponentiation, and the constant one, for $\times$, $\to$ and $\mathbf{T}$) was proved to be complete for the Cartesian closed categories [19, 122]. Since the models of the lambda calculus with unit, product- and function-types are exactly the Cartesian closed categories [19], the set is also complete for the type isomorphisms we examine. Through the Curry-Howard isomorphism [79], these isomorphisms are also equivalent to equational equality in positive intuitionistic logic so the same axioms apply there too (again, with appropriate notational changes).

Besides their theoretical connections, type isomorphisms can be used as a means of searching large program libraries. Specifically, the desired type of a function is used as a search key and functions with isomorphic types are returned as candidates. A famous example [116] shows that even the simple function, folding a list, can be implemented with many different types, varying argument order and the use of “Curried” style. Employing type isomorphisms in the search will retrieve all compatible function implementations. Moreover, the isomorphism proof can often automatically generate bridge code converting the functions found to the desired type. It was even argued [10] that type isomorphisms can be employed in proof reuse.

Second order isomorphisms augment first order isomorphisms with universal quantifiers, as in $\forall A.A \to A = \forall B.B \to B$. Universal quantifiers make second order isomorphisms more effective in searching program libraries since they are necessary to capture parametric polymorphism. While some of the issues of second order isomorphisms are similar (some of the space sharing techniques are applicable), they are known to be graph isomorphism complete [11, 39] and we do not attempt to decide them in this work. A different system of type isomorphisms is that of the core ML language. It is known [38] that second order isomorphisms are insufficient to describe these, although the addition of one more axiom suffices.

Recursive variants of the type isomorphism problem at our hand were also considered in the literature. In the Mockingbird project the recursive type system comprised of product- and function-types [8, 9, 111]. Gil [67] describes how algorithms for polynomial equality can be used for deciding isomorphism in the "algebraic type system", i.e., the recursive type system comprising of union- and product-types.

The more general isomorphism problem, for a non-recursive type system which includes product-, union- *and* function-types is equivalent to Tarski's *high school algebra problem* [125]. Such a system does not have a finite and complete set of axioms. Nonetheless, there exists a (non-polynomial) algorithm for determining isomorphism [73]. There also exists a (non-polynomial) algorithm for deciding isomorphism in the recursive "algebraic type system" [67]. Finally, we should mention that adding empty and sum types breaks down the relationship between the equational theory and type isomorphisms [63].

outline The first order isomorphism problem and its variants are defined in Section 7.1. Section 7.2 gives the intuition for solving this problem. More specifically, this section describes how previous work used reduction systems in order to obtain normal forms which are more easily compared. Sections 7.3–7.9 present different stages of our main algorithm for the first order isomorphism problem. The pieces are then put together in Section 7.10. Finally, Section 7.11 mentions some open problems and directions for future research.

7.1 Definitions: The First Order Isomorphism Problem and Its Variants

Here we concentrate on first order isomorphism and two restricted variants (product and linear isomorphism). We now make the necessary definitions in order to give a precise statement of the problem and its variants.

Next we define four successive theories of isomorphism of types.

Definition 7.1 *Let* Equality *be the theory of equality of types defined as the set of propositions obtained by the deductive closure of the axiom schema*

$(\mathcal{A}.0)\qquad A = A$ *(Reflexive)* *and the following four inference rules.*

$$\frac{A = B}{B = A} \qquad \textit{symmetry}$$

$$\frac{A = B, B = C}{A = C} \qquad \textit{transitivity}$$

$$\frac{A = B, C = D}{A \times C = B \times D} \qquad \textit{congruence of } \times$$

$$\frac{A = B, C = D}{A \to C = B \to D} \qquad \textit{congruence of } \to$$

Thus, Equality is the usual theory of equality, sometimes denoted as Th^0 [32].

Definition 7.2 *Let* Product *be the theory* Equality *augmented with axiom schemas* $\mathcal{A}.1$*–*$\mathcal{A}.5$*. Let* Linear *be the theory* Product *augmented with axiom schema* $\mathcal{A}.6$*. Let* First *be the theory* Linear *augmented with axiom schema* $\mathcal{A}.7$*.*

Theory Product adds the unit axioms to the theory of equality as well as the rules of commutative and associative products. The currying axiom is added in theory Linear. Finally, First is the theory of first order isomorphisms, which is often referred to in the literature as $Th^1_{\times T}$ [19, 20, 39].

When $\mathbf{T}$ does not occur in the input, it is convenient to use theory variants which do not include the unit axioms.

Definition 7.3 *Let* Product$^-$ *be the theory* Equality *augmented with axiom schemas* $\mathcal{A}.4$ *and* $\mathcal{A}.5$*. Let* Linear$^-$ *be the theory* Product$^-$ *augmented with axiom schema* $\mathcal{A}.6$*. Let* First$^-$ *be the theory* Linear$^-$ *augmented with axiom schema* $\mathcal{A}.7$*.*

Definition 7.4 (Axiom instance) *An* instance of an axiom $\mathcal{A}$ *is the result of a consistent substitution of all the variables in* $\mathcal{A}$ *by type expressions of the general grammar.*

For example, $\big(\mathsf{a} \to (\mathbf{T} \times \mathsf{b})\big) \times \mathsf{c} = \mathsf{c} \times \big(\mathsf{a} \to (\mathbf{T} \times \mathsf{b})\big)$ is an instance of the commutative axiom $\mathcal{A}.4$.

Definition 7.5 (Derivation sequence) *Let* Θ *be a theory, e.g.,* $\Theta =$ Equality*, or* $\Theta =$ First$^-$*. Then, the sequence* $\tau_1 = \tau_1', \ldots, \tau_m = \tau_m'$ *is called a* derivation sequence *in* Θ *if for* $i = 1, \ldots, m$*,* $\tau_i = \tau_i'$ *is either an instance of an axiom in* Θ *or the result of applying one of the four inference rules on previous equalities. For types* τ, τ' *we write* $\Theta \vdash \tau = \tau'$ *when there exists a derivation sequence ending with the equality* $\tau = \tau'$*.*

Let τ and τ' be two given types. We use the notation $\tau = \tau'$ as an abbreviation for Equality $\vdash \tau = \tau'$.

Definition 7.6 *The first order isomorphism problem is to decide whether* First $\vdash \tau = \tau'$*.*

The first order isomorphism problem has been known to be decidable for over a decade [19, 122]. Previous to our work, the best known bound was $O(n^2 \log n)$ time using $O(n^2)$ space [32]. *Our main result is in reducing the time to $O(n \log^2 n)$ time and the space to $O(n)$.*

One of the difficult issues in obtaining an efficient algorithm for the problem is dealing with the commutative and associative nature of product (axioms $\mathcal{A}.4$ and $\mathcal{A}.5$). Concentrating on this we define the product isomorphism problem.

Definition 7.7 *The product isomorphism problem is to decide whether* $\mathsf{Product} \vdash \tau = \tau'$.

We apply the standard abbreviation of using the $\prod$ symbol to denote (an associated to the left) product of several *terms*, i.e., for $k \geq 2$,

$$\prod_{i=1}^{k} \tau_i = \left(\cdots \left((\tau_1 \times \tau_2) \times \tau_3 \right) \cdots \times \tau_k \right), \tag{7.2}$$

When the commutative and associative axioms apply, we shall write products without parenthesis. Consider, for example, the following product:

$$\mathsf{abracadabra}. \tag{7.3}$$

(Lower case, sanserif letters denote here and henceforth primitive-types. We shall use the arithmetical and type notations interchangeably. No confusion will arise.) An instance of the product isomorphism problem variant is to determine whether the above is isomorphic to

$$\mathsf{carrabadaba}. \tag{7.4}$$

One may be tempted to attack the problem by bringing each product into a unique sorted normal form, which in this case is

$$\mathsf{aaaaabbcdrr}. \tag{7.5}$$

In this chapter we show that the product isomorphism problem is decidable in linear time.[2] This result is based on the observation that it can be determined that (7.3) and (7.4) are isomorphic without using a super-linear sorting procedure, but rather by employing an algorithm for *multi-set comparison*. More generally, to determine whether $\prod_{i=1}^{k} A_i$ is isomorphic to $\prod_{i=1}^{k} B_k$ the multi-set comparison algorithm checks whether there exists a permutation π such that $A_{\pi(i)}$ is isomorphic to B_i.

This product isomorphism variant was not considered previously as such in the literature. Palsberg and Zhao [111] gave an $O(n^2)$ time algorithm for a *recursive* product isomorphism problem, defined by the addition of a grammar rule $\tau ::= \mu\alpha.\tau$ where α is a type variable, and a folding/unfolding axiom

$$(\mathcal{A}.8) \qquad \mu\alpha.A = A[(\mu\alpha.A)/\alpha].$$

[2]Jha (personal communication, September 2002) reports on independent discovery of an algorithm for this sub-problem, with similar complexity bounds, published in [81].

(As usual, the notation $A[B/\alpha]$ stands for a type expression A where each occurrence of α is replaced by B.) This result was later improved to $O(n \log n)$ time [80] using a reduction to the problem of finding size-stable partitions of a directed graph.

We note that the recursive product isomorphism problem is not a simple a generalization of our product isomorphism problem. The reason is that isomorphism between recursive product-types should be defined in terms of their infinite unfoldings which are regular trees. To reason about these infinite structure, inductive variants of the *congruence of* $\times$ and *congruence of* $\rightarrow$ inference rules must be used. It was found (Palsberg, personal communication) that the combination of these variants with the folding/unfolding axiom and the unit axioms $\mathcal{A}.1$–$\mathcal{A}.3$. gives rise to an inconsistent system. These axioms were therefore omitted from the recursive product type systems. It remains a challenge to find a reformulation of the inference rules in Definition 7.1 which is consistent with all axioms $\mathcal{A}.1$–$\mathcal{A}.8$.

More difficult than the product isomorphism problem is the problem variant defined by the Linear theory, which adds the currying axiom.

Definition 7.8 *The linear isomorphism problem is to decide whether* $\mathsf{Linear} \vdash \tau = \tau'$.

Polynomial time results for this problem were known before those of the first order problem. Linear isomorphism can be decided in linear space and $O(n \log^2 n)$ time [6]. Although not previously mentioned, both algorithms [32, 80] improve the running time to $O(n \log n)$. *We advance the state of the art by showing that linear isomorphism is also decidable in linear time.*

Linear isomorphism combined with the folding/unfolding axiom may generate products with an unbounded number of terms, which makes it difficult to apply the standard algorithms for recursive type isomorphisms. Consider, for example, the type

$$\mu\alpha.(\mathsf{a} \rightarrow \alpha). \tag{7.6}$$

The following equality is an instance of the folding/unfolding axiom

$$\mu\alpha.(\mathsf{a} \rightarrow \alpha) = \mathsf{a} \rightarrow \big(\mu\alpha.(\mathsf{a} \rightarrow \alpha)\big).$$

Repeated use of the folding/unfolding axiom proves that type (7.6) is isomorphic to

$$\mathsf{a} \rightarrow \Big(\mathsf{a} \rightarrow \cdots \rightarrow \big(\mu\alpha.(\mathsf{a} \rightarrow \alpha)\big) \cdots\Big).$$

Finally, by using the currying axiom we can produce a product with any number of terms.

The final step toward solving the first order isomorphism problem is to deal with the distributive axiom $\mathcal{A}.7$. As we shall see, the difficulty in doing so is that a naive application of this axiom may lead to an exponential blowup of the input types.

7.2 Intuition: Reduction Systems and Normal Forms

Isomorphism proofs are usually based upon *reduction systems* producing a normal form representation of the input, which can be more easily compared. We assume that types use a standard expression-tree representation in memory, and that each *rule application* in the reduction system is implemented as a transformation of this data structure.

For example, the reduction system of Rittri [116] has seven rules

$$\begin{array}{lrcl} \mathcal{R}.1 & \mathbf{T}\times A & \Rightarrow & A \\ \mathcal{R}.2 & A\times\mathbf{T} & \Rightarrow & A \\ \mathcal{R}.3 & \mathbf{T}\to A & \Rightarrow & A \\ \mathcal{R}.4 & A\to\mathbf{T} & \Rightarrow & \mathbf{T} \\ \mathcal{R}.5 & A\times(B\times C) & \Rightarrow & (A\times B)\times C \\ \mathcal{R}.6 & A\to(B\to C) & \Rightarrow & (A\times B)\to C \\ \mathcal{R}.7 & A\to(B\times C) & \Rightarrow & (A\to B)\times(A\to C) \end{array} \tag{7.7}$$

Rittri proved that the rules $\mathcal{R}.1$–$\mathcal{R}.7$ are confluent and terminating. Therefore, by repeated application of the rules the input types are reduced to a *normal form*.

In the degenerate case in which one or both of the inputs is reduced to $\mathbf{T}$, the input types are isomorphic if and only if they both reduce to $\mathbf{T}$. (This intuitive statement is given a formal proof in Section 7.3.) Otherwise, the normal forms do not contain the symbol $\mathbf{T}$. Furthermore, these rules can always simplify the structure of the right operand of $\to$, unless it is a primitive-type.

An algorithm for deciding first order isomorphism is to recursively compare the resulting normal forms: two nodes are isomorphic if they are of the same kind (product or function) and their operands are isomorphic. In function-nodes the comparison of arguments is straightforward: the left (right) operand of one node must be isomorphic to the left (right) operand of the other. In comparing product-nodes however we must solve an instance of the product polymorphism problem to check whether the terms of one node is pair-wise isomorphic to some permutation of the terms of the other node. If this comparison is not done carefully it adds to the complexity of the problem.

An even more serious inefficiency factor is that the system (7.7) (specifically, the distributive rule $\mathcal{R}.7$) may introduce an exponential blowup in the size of the representation. Rules $\mathcal{R}.1$–$\mathcal{R}.6$ do not increase the representation size. However, each application of $\mathcal{R}.7$ creates a duplicate copy of the subtree whose root is A. Repeated applications may produce a very large normal form representation. In the sequence of types $\{X_i\}$, defined by $X_0 = \mathsf{a}$ and $X_i = (\mathsf{b}_i\mathsf{c}_i)^{X_{i-1}}$ for $i > 0$, we have that $X_n \Rightarrow \mathsf{b}_n^{X_{n-1}}\mathsf{c}_i^{X_{i-1}}$ and successive applications of this rule to each occurrence of X_i, $i = n-1, \ldots, 1$, will lead to exponentially many copies of a in the normal form of X_n.

If graphs, rather than trees, are used to represent types, then an application of $\mathcal{R}.7$, can be implemented by *sharing* the node representing A. This sharing can be thought of as an application of a slightly different transformation

$$A \to (B\times C) \Rightarrow \begin{cases} (\alpha\to B)\times(\alpha\to C) \\ \alpha = A \end{cases}, \tag{7.8}$$

where a newly introduced symbolic variable α is represented as a pointer to the data-structure representation of type A.

Rittri [117] observed that using (7.8) ensures a polynomially sized representation of the normal form: Each application of transformation (7.8) adds one edge to the graph.

The application reduces the nesting level of the $\times$ node, and this nesting level cannot be increased by the other rules. We obtain that the space of the graph normal form is $O(n^2)$ by noticing that initially there are at most n product-nodes, and that even though additional product-nodes may be created by $\mathcal{R}.6$, these nodes cannot take part in the other two rules.

To see that the representation can indeed by quadratic, consider the following example (written using the arithmetical notation):

$$\left(\mathsf{b}_1\left(\mathsf{b}_2\cdots\left(\mathsf{b}_{n-2}(\mathsf{b}_{n-1}\mathsf{b}_n^{\mathsf{a}_n})^{\mathsf{a}_{n-1}}\right)^{\mathsf{a}_{n-2}}\cdots\right)^{\mathsf{a}_2}\right)^{\mathsf{a}_1}, \tag{7.9}$$

whose normal form is

$$\mathsf{b}_1^{\mathsf{a}_1}\mathsf{b}_2^{\mathsf{a}_2\mathsf{a}_1}\cdots\mathsf{b}_{n-1}^{\mathsf{a}_{n-1}\cdots\mathsf{a}_1}\mathsf{b}_n^{\mathsf{a}_n\cdots\mathsf{a}_1}. \tag{7.10}$$

This normal form consumes quadratic space if derived by applying $\mathcal{R}.7$ starting at the inner most parenthesis.

Remark 7.9 *Deriving* (7.9) *starting at the outer-most parenthesis, yields the representation*

$$\mathsf{b}_1^{\alpha_1}\cdots\mathsf{b}_n^{\alpha_n}, \tag{7.11}$$

where $\alpha_1 = \mathsf{a}_1$*, and* $\alpha_i = \mathsf{a}_i\alpha_{i-1}$ *for* $i = 2, \ldots, n$*. Note that* (7.11) *requires only linear space whereas* (7.10) *is quadratic.*

Having bounded the space explosion, Rittri stopped short of giving a polynomial time algorithm for the problem. By noticing that the graph representation is acyclic, and by using a variant of Rittri's normal form, Considine [32] was able to reduce the runtime to polynomial. We should note that Considine's rules were different than Rittri's in that rule $\mathcal{R}.6$ was applied in the opposite direction. The resulting normal form is such that instead of A^{BCD}, it uses the equivalent representation $\left(\left(A^B\right)^C\right)^D$. Thus, strictly speaking, his normal form did not use product-nodes, other than in the upper most level. However, the alternative representation must still deal with the difficulties of associativity and commutativity as in the more familiar representation of products.

Considine's algorithm partitions all nodes in the directed acyclic graph (DAG) representation of the input types into equivalence classes, such that all nodes in the same equivalence class are isomorphic. This partitioning is built in a bottom-up traversal of the DAGs, while maintaining a hash table mapping each node into the unique identifier of its equivalence class. The most difficult task in this traversal was to determine whether product-nodes are isomorphic. Two key properties made Considine's $O(n^2 \log n)$ time and $O(n^2)$ space result possible:

1. *Expansion of product-types.* Considine showed that his normal form, which includes complete expansion of product-types, is such that each product consists of no more than n terms.

2. *Sorting product terms.* Since the graph is acyclic, terms in product-types must have been visited and classified by the bottom up traversal before the product itself. Each product-node is first normalized by sorting the identifiers of the equivalence classes of their terms. The fact that the order of terms is completely determined by this sorting makes it possible to employ a *hash-consing* technique to produce a unique identifier for each product-type, thereby partitioning product-type nodes into equivalence classes.

Our algorithm uses the same bottom-up classification of nodes into equivalence classes. However, the reduction of space to $O(n)$ and of time to $O(n \log^2 n)$ are made possible by breaking away from the above principles. Specifically, the new algorithm is characterized by:

1. *Application of $\mathcal{R}.7$ to "outer-most" functions first.* As demonstrated in Remark 7.9 the space is kept linear if the distributive rule is applied starting at the outer-most parenthesis.

2. *Unexpanded product-types.* The expansion of product-types leads to quadratic time and space. Instead, we describe a graph based representation, which keeps the space linear, and show that unexpanded products can still be efficiently compared.

3. *Unsorted product terms.* Isomorphism of product-nodes is decided by a procedure which can be thought of as hashing or range compaction, rather than sorting. A similar procedure is used to partition the multi-sets of products in each stage of the traversal into their equivalence classes.

Road map Our algorithms employ four successive normal forms, all of which can be computed in linear time and space. Each normal form stands for a "simpler" isomorphic representation, obtained by exhaustively applying some of the rules (7.7).

The normal form $\mathrm{nf}_{\mathbf{T}}$, described in Section 7.3, is computed by applying rules $\mathcal{R}.1$–$\mathcal{R}.4$ to remove (essentially) all occurrences of $\mathbf{T}$. We further show in this section, that $\mathrm{nf}_{\mathbf{T}}$ makes it possible to completely ignore the unit axioms in the main algorithms.

The normal form nf_{c}, which takes care of the *currying* axiom, is the subject of Section 7.4, where we show how linear isomorphism can be reduced to product isomorphism.

To solve the product isomorphism problem, we need a procedure for comparing long products without sorting their terms. Section 7.5 develops this procedure as part of a general algorithm for multi-set partitioning. Section 7.6 then gives the concrete algorithm for the product isomorphism problem. In the algorithm the *associative* rule $\mathcal{R}.5$ is first applied to produce the normal form nf_{a}. The normalized types are then compared in a bottom-up traversal, while invoking the multi-set partitioning algorithm at each level.

Section 7.7 then shows how an exhaustive application of the *distributive* rule $\mathcal{R}.7$ produces the normal form nf_{d}. A linear space encoding for nf_{d}, called the $\mathbf{P}/\mathbf{F}$-graph, is also described in this section. Unexpanded products in the $\mathbf{P}/\mathbf{F}$-graph form a *tree structure*, such that each product inherits the terms of its parent. Section 7.8 employs multi-set partitioning in comparing unexpanded products in this tree structure. Section 7.9 fine-tunes this procedure to its application in a bottom-up classification of the nodes of the $\mathbf{P}/\mathbf{F}$-graph. Finally, we present our main algorithm for deciding first order isomorphisms of simple types in Section 7.10. Section 7.11 lists some open questions.

7.3 Eliminating Unit Types

This section describes a linear time and space algorithm for eliminating the unit axioms. Algorithm `EliminateUnits` receives as input two types: τ and τ', both conforming to the *general grammar*, describing arbitrary first order types.

General Grammar

$$\tau ::= \mathbf{T} \quad | \quad x \quad | \quad \tau \to \tau \quad | \quad \tau \times \tau.$$

The output comprises two types σ and σ', such that

$$\mathsf{First} \vdash \tau = \tau' \Leftrightarrow \mathsf{First}^{-} \vdash \sigma = \sigma'.$$

(At the end of this section we show that a similar claim can be made for theories Linear and Product.) The details are in Algorithm 7.1.

Algorithm 7.1 `EliminateUnits` (τ, τ')

Given two types τ and τ' conforming to the general grammar, return either (i) a decision whether First $\vdash \tau = \tau'$*, or (ii) two types σ, σ' conforming to the no-unit grammar such that* First $\vdash \tau = \tau' \Leftrightarrow$ First$^{-} \vdash \sigma = \sigma'$*.*

```
1: $\sigma \leftarrow \mathrm{nf}_{\mathbf{T}}(\tau)$
2: $\sigma' \leftarrow \mathrm{nf}_{\mathbf{T}}(\tau')$
3: If $\sigma = \mathbf{T}$ and $\sigma' = \mathbf{T}$ then
4:    return true // Types $\tau$ and $\tau'$ are isomorphic
5: else if $\sigma = \mathbf{T}$ or $\sigma' = \mathbf{T}$ then
6:    return false // Types $\tau$ and $\tau'$ are not isomorphic
7: else
8:    return $\langle \sigma, \sigma' \rangle$
9: fi
```

If either of τ or τ' is isomorphic to $\mathbf{T}$ then the algorithm returns a decision whether First $\vdash \tau = \tau'$ (lines 4 and 6). Otherwise, i.e., when both τ and τ' are not isomorphic to $\mathbf{T}$, the algorithm returns two types σ and σ' such that First $\vdash \tau = \tau' \Leftrightarrow$ First$^{-} \vdash \sigma = \sigma'$ (line 8). Both σ and σ' conform to the following *no-unit grammar*, in which the symbol $\mathbf{T}$ never occurs.

No-Unit Grammar

$$\tau ::= x \quad | \quad \tau \to \tau \quad | \quad \tau \times \tau$$

The crux of the algorithm is the transformation of the inputs into their normal form in lines 1 and 2. For a type τ, its normal form $\mathrm{nf}_{\mathbf{T}}(\tau)$ is a type isomorphic to τ, i.e., First $\vdash \tau = \mathrm{nf}_{\mathbf{T}}(\tau)$, where $\mathrm{nf}_{\mathbf{T}}(\tau)$ is either the type $\mathbf{T}$ or it conforms to the no-unit grammar.

The following is an algorithmic definition of the normalizing function $\mathrm{nf}_{\mathbf{T}}$.[3] The function recursively traverses the tree representing the input type, while applying rules $\mathcal{R}.1$–$\mathcal{R}.4$ whenever possible.

$$\mathrm{nf}_{\mathbf{T}}(\tau) = \begin{cases} \mathbf{T} & \text{if } \tau = \mathbf{T} \\ x & \text{if } \tau = x \\ R_{1,2}(\mathrm{nf}_{\mathbf{T}}(\tau_a), \mathrm{nf}_{\mathbf{T}}(\tau_b)) & \text{if } \tau = \tau_a \times \tau_b \\ R_{3,4}(\mathrm{nf}_{\mathbf{T}}(\tau_a), \mathrm{nf}_{\mathbf{T}}(\tau_b)) & \text{if } \tau = \tau_a \rightarrow \tau_b \end{cases} \tag{7.12}$$

After the children of a node have been simplified by the recursive calls, function $\mathrm{nf}_{\mathbf{T}}$ may invoke, depending on the node type, one of two auxiliary functions to simplify the node itself. The first such function applies the product-unit rules ($\mathcal{R}.1$ and $\mathcal{R}.2$).

$$R_{1,2}(\sigma_a, \sigma_b) = \begin{cases} \sigma_b & \text{if } \sigma_a = \mathbf{T} & \textit{// apply rule } \mathcal{R}.1 \\ \sigma_a & \text{if } \sigma_b = \mathbf{T} & \textit{// apply rule } \mathcal{R}.2 \\ \sigma_a \times \sigma_b & \text{otherwise} \end{cases} \tag{7.13}$$

The other auxiliary function applies the function-unit rules ($\mathcal{R}.3$ and $\mathcal{R}.4$).

$$R_{3,4}(\sigma_a, \sigma_b) = \begin{cases} \sigma_b & \text{if } \sigma_a = \mathbf{T} & \textit{// apply rule } \mathcal{R}.3 \\ \mathbf{T} & \text{if } \sigma_b = \mathbf{T} & \textit{// apply rule } \mathcal{R}.4 \\ \sigma_a \rightarrow \sigma_b & \text{otherwise} \end{cases} \tag{7.14}$$

Let $|\tau|$ denote the *size* of a type τ, defined as the number of nodes in the standard abstract syntax tree representation of τ. Many of our proofs employ *structural induction* which is essentially induction on the input size. In the inductive step, we shall rely on the *type decomposability property*: if $|\tau| > 1$ (i.e., $\tau \neq x$ and $\tau \neq \mathbf{T}$) then τ is represented as a type-operator node with two children representing types τ_a and τ_b, such that $|\tau| = |\tau_a| + |\tau_b| + 1$.

Lemma 7.10 *Let τ be a type which conforms to the general grammar, and let $\sigma = \mathrm{nf}_{\mathbf{T}}(\tau)$. Then,* **(i)** *the invocation $\mathrm{nf}_{\mathbf{T}}(\tau)$ requires $O(|\tau|)$ time,* **(ii)** $|\sigma| \leq |\tau|$, **(iii)** $\sigma = \mathbf{T}$ *or σ conforms to the no-unit grammar, and* **(iv)** Product $\vdash \tau = \sigma$.

PROOF. All parts are proved by structural induction. The inductive base, $|\tau| = 1$, is covered by the first two cases ($\tau = x$ and $\tau = \mathbf{T}$) in (7.12). Both these cases execute in constant time, and their output is identical to their input. Moreover, this output either conforms to the no-unit grammar or is $\mathbf{T}$.

[3] Here and henceforth, we use the same notation for the *normal form*, and for the (algorithmic) function which given a type, generates and returns its normal form. No confusion should arise as a result of this overloading.

In proving the inductive step we use the inductive hypothesis and the decomposability property. For **(i)** we note that only a constant amount of work is carried out prior to and after the recursive calls (i.e., in $R_{1,2}$ and $R_{3,4}$). Noting that $R_{1,2}$ and $R_{3,4}$ do not create new nodes proves the inductive step of **(ii)**. The inductive step of **(iii)** is carried out by checking that the output of $R_{1,2}$ and $R_{3,4}$ satisfies **(iii)** whenever their input does. Part **(iv)** is proved by noting that functions $R_{1,2}$ and $R_{3,4}$ only apply rules conforming to the axioms $\mathcal{A}.1$–$\mathcal{A}.4$. $\square$

Lemma 7.10 proves the correctness of Algorithm 7.1 in the cases it terminates in line 4. Next we would like to prove that when the algorithm terminates in line 6 then τ and τ' are indeed not isomorphic. Note that the algorithm terminates in line 6 if and only if either $\sigma = \mathbf{T}$ and $\sigma' \neq \mathbf{T}$ or the reverse. Therefore we must prove that $\mathbf{T}$ cannot be isomorphic to any type σ which conforms to the no-unit grammar. We will use the technique of abstract interpretation [35] for doing so.

For a type τ define the abstract interpretation function $\mathrm{is}_{\mathbf{T}}(\tau)$ as follows

$$\mathrm{is}_{\mathbf{T}}(\tau) = \begin{cases} 1 & \text{if } \tau = \mathbf{T} \\ 0 & \text{if } \tau = x \\ \mathrm{is}_{\mathbf{T}}(\tau_a) \cdot \mathrm{is}_{\mathbf{T}}(\tau_b) & \text{if } \tau = \tau_a \times \tau_b \\ \mathrm{is}_{\mathbf{T}}(\tau_b) & \text{if } \tau = \tau_a \to \tau_b \end{cases} \tag{7.15}$$

Note that $\mathrm{is}_{\mathbf{T}}(\tau)$ returns either 0 or 1. We next prove that $\mathrm{is}_{\mathbf{T}}(\tau)$ is 1 precisely when $\mathrm{nf}_{\mathbf{T}}(\tau) = \mathbf{T}$ (hence the name $\mathrm{is}_{\mathbf{T}}$).

Lemma 7.11 $\mathrm{nf}_{\mathbf{T}}(\tau) = \mathbf{T} \Leftrightarrow \mathrm{is}_{\mathbf{T}}(\tau) = 1.$

PROOF. By examining the definitions of $\mathrm{nf}_{\mathbf{T}}$, $R_{1,2}$ and $R_{3,4}$ we see that $\mathrm{nf}_{\mathbf{T}}(\tau) = \mathbf{T}$ if and only if one of the following holds

1. $\tau = \mathbf{T}$.
2. $\tau = \tau_a \times \tau_b$, where $\mathrm{nf}_{\mathbf{T}}(\tau_a) = \mathbf{T}$ and $\mathrm{nf}_{\mathbf{T}}(\tau_b) = \mathbf{T}$.
3. $\tau = \tau_a \to \tau_b$, where $\mathrm{nf}_{\mathbf{T}}(\tau_b) = \mathbf{T}$.

Therefore $\mathrm{nf}_{\mathbf{T}}(\tau) = \mathbf{T}$ if and only if $\mathrm{is}_{\mathbf{T}}(\tau) = 1$. $\square$

Lemma 7.12 First $\vdash \tau = \tau' \Rightarrow \mathrm{is}_{\mathbf{T}}(\tau) = \mathrm{is}_{\mathbf{T}}(\tau')$

PROOF. By induction on the length of the derivation sequence of First $\vdash \tau = \tau'$. Recall that each equality in the derivation sequence is either an instance of an axiom or an application of one of the inference rules on previous equalities.

The induction base is that there is precisely one such equality $\tau = \tau'$, which must be an instance of an axiom $\mathcal{A}.0, \ldots, \mathcal{A}.7$. We can easily check in each of the axioms that $\mathrm{is}_{\mathbf{T}}(\tau) = \mathrm{is}_{\mathbf{T}}(\tau')$. For example, if $\tau = \tau'$ is an instance of $\mathcal{A}.7$. then $\tau = \tau_a \to (\tau_b \times \tau_c)$ and $\tau' = (\tau_a \to \tau_b) \times (\tau_a \to \tau_c)$. We have

$$\mathrm{is}_{\mathbf{T}}(\tau) = \mathrm{is}_{\mathbf{T}}\big(\tau_a \to (\tau_b \times \tau_c)\big) = \mathrm{is}_{\mathbf{T}}(\tau_b \times \tau_c) = \mathrm{is}_{\mathbf{T}}(\tau_b) \cdot \mathrm{is}_{\mathbf{T}}(\tau_c),$$

and

$$\mathrm{is}_{\mathbf{T}}(\tau') = \mathrm{is}_{\mathbf{T}}\big((\tau_a \to \tau_b) \times (\tau_a \to \tau_c)\big) = \mathrm{is}_{\mathbf{T}}(\tau_a \to \tau_b) \cdot \mathrm{is}_{\mathbf{T}}(\tau_a \to \tau_c) = \mathrm{is}_{\mathbf{T}}(\tau_b) \cdot \mathrm{is}_{\mathbf{T}}(\tau_c).$$

To prove the induction step we examine the last step of the derivation sequence. If this step is an axiom instance, then the same considerations as in the induction base apply. Otherwise one of the following inference rules was applied: symmetry, transitivity, congruence of $\times$, or congruence of $\to$. We can easily check each of inference rules by using the inductive hypothesis. For instance, suppose that the congruence rule of $\times$ was applied:

$$\frac{\tau_a = \tau_b, \tau_c = \tau_d}{\tau_a \times \tau_c = \tau_b \times \tau_d}.$$

By the inductive hypothesis, we have that $\mathrm{is}_{\mathbf{T}}(\tau_a) = \mathrm{is}_{\mathbf{T}}(\tau_b)$ and $\mathrm{is}_{\mathbf{T}}(\tau_c) = \mathrm{is}_{\mathbf{T}}(\tau_d)$. Therefore, we can deduce that

$$\mathrm{is}_{\mathbf{T}}(\tau_a \times \tau_c) = \mathrm{is}_{\mathbf{T}}(\tau_a) \cdot \mathrm{is}_{\mathbf{T}}(\tau_c) = \mathrm{is}_{\mathbf{T}}(\tau_b) \cdot \mathrm{is}_{\mathbf{T}}(\tau_d) = \mathrm{is}_{\mathbf{T}}(\tau_b \times \tau_d).$$

□

Corollary 7.13 *Let σ be a type conforming to the no-unit grammar. Then σ is not isomorphic to $\mathbf{T}$, i.e.,* First $\nvdash \sigma = \mathbf{T}$.

PROOF. Assume by contradiction that First $\vdash \sigma = \mathbf{T}$. Then, by Lemma 7.12, $\mathrm{is}_{\mathbf{T}}(\sigma) = \mathrm{is}_{\mathbf{T}}(\mathbf{T})$. Since σ conforms to the no-unit grammar, we have that $\mathrm{is}_{\mathbf{T}}(\sigma) = 0$, which contradicts the fact that $\mathrm{is}_{\mathbf{T}}(\mathbf{T}) = 1$. □

Finally, we will prove the correctness of Algorithm 7.1 in the cases it terminates in line 8, i.e., we need to show that

$$\text{First} \vdash \tau = \tau' \Leftrightarrow \text{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau').$$

The $\Leftarrow$ direction follows directly from Lemma 7.10**(iv)** combined with the facts that First$^-$ $\subseteq$ First and Product $\subseteq$ First.

Lemma 7.14 *Let τ and τ' be two types conforming to the general grammar. Then,*

$$\text{First} \vdash \tau = \tau' \Rightarrow \text{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau').$$

PROOF. By induction on the length of the derivation sequence of First $\vdash \tau = \tau'$, whose final step must be the equality $\tau = \tau'$. In the induction base, this equality must be instance of one of the axioms $\mathcal{A}.0, \ldots, \mathcal{A}.7$. If $\tau = \tau'$ is an instance of $\mathcal{A}.3$, then $\tau = \mathbf{T} \to \tau_a$

and $\tau' = \tau_a$. We see that $\mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau')$, and hence $\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau')$. A similar consideration and conclusion applies if $\tau = \tau'$ is an instance of axioms $\mathcal{A}.0$–$\mathcal{A}.2$

Suppose that $\tau = \tau'$ is an instance of the commutative axiom $\mathcal{A}.4$, i.e., $\tau = \tau_a \times \tau_b$ and $\tau' = \tau_b \times \tau_a$. We have

$$\begin{aligned}\mathrm{nf}_{\mathbf{T}}(\tau) &= R_{1,2}(\mathrm{nf}_{\mathbf{T}}(\tau_a), \mathrm{nf}_{\mathbf{T}}(\tau_b)),\\ \mathrm{nf}_{\mathbf{T}}(\tau') &= R_{1,2}(\mathrm{nf}_{\mathbf{T}}(\tau_b), \mathrm{nf}_{\mathbf{T}}(\tau_a)).\end{aligned}$$

If either $\mathrm{nf}_{\mathbf{T}}(\tau_a) = \mathbf{T}$ or $\mathrm{nf}_{\mathbf{T}}(\tau_b) = \mathbf{T}$ then $\mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau')$, therefore $\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau')$. Otherwise

$$\begin{aligned}\mathrm{nf}_{\mathbf{T}}(\tau) &= \mathrm{nf}_{\mathbf{T}}(\tau_a) \times \mathrm{nf}_{\mathbf{T}}(\tau_b),\\ \mathrm{nf}_{\mathbf{T}}(\tau') &= \mathrm{nf}_{\mathbf{T}}(\tau_b) \times \mathrm{nf}_{\mathbf{T}}(\tau_a),\end{aligned}$$

and the commutative axiom $\mathcal{A}.4$ proves that $\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau')$. A similar, though more laborious, consideration proves the same induction base in the case that $\tau = \tau'$ is an instance of $\mathcal{A}.5$–$\mathcal{A}.7$.

In the induction step, we focus on the case that the final equality was obtained by one of the inference rules: symmetry, transitivity, congruence of $\times$, or congruence of $\rightarrow$. (The case that this equality is an axiom instance is identical to the induction base.)

Consider, for instance, the inference rule for congruence of $\times$. Then $\tau = \tau_a \times \tau_b$ and $\tau' = \tau_c \times \tau_d$. The inductive hypothesis is that $\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau_a) = \mathrm{nf}_{\mathbf{T}}(\tau_c)$ and $\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau_b) = \mathrm{nf}_{\mathbf{T}}(\tau_d)$. We need to show that $\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau_a \times \tau_b) = \mathrm{nf}_{\mathbf{T}}(\tau_c \times \tau_d)$, or in other words, that

$$\mathsf{First}^- \vdash R_{1,2}\big(\mathrm{nf}_{\mathbf{T}}(\tau_a), \mathrm{nf}_{\mathbf{T}}(\tau_b)\big) = R_{1,2}\big(\mathrm{nf}_{\mathbf{T}}(\tau_c), \mathrm{nf}_{\mathbf{T}}(\tau_d)\big). \tag{7.16}$$

Examining definition (7.13) of $R_{1,2}$ we see that the proof must distinguish between several cases, depending on whether the arguments to this function are $\mathbf{T}$.

To make this distinction, we apply Lemma 7.12, obtaining that $\mathrm{nf}_{\mathbf{T}}(\tau_a) = \mathbf{T}$ if and only if $\mathrm{nf}_{\mathbf{T}}(\tau_c) = \mathbf{T}$, and $\mathrm{nf}_{\mathbf{T}}(\tau_b) = \mathbf{T}$ if and only if $\mathrm{nf}_{\mathbf{T}}(\tau_d) = \mathbf{T}$. (The lemma condition is met by the inductive hypothesis and the fact that $\mathsf{First}^- \subseteq \mathsf{First}$.)

Consider the case that $\mathrm{nf}_{\mathbf{T}}(\tau_a) \neq \mathbf{T}$ and $\mathrm{nf}_{\mathbf{T}}(\tau_b) \neq \mathbf{T}$. Then, (7.16) takes the form

$$\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau_a) \times \mathrm{nf}_{\mathbf{T}}(\tau_b) = \mathrm{nf}_{\mathbf{T}}(\tau_c) \times \mathrm{nf}_{\mathbf{T}}(\tau_d).$$

The derivation sequence for this can be obtained by concatenating the derivation sequences of the inductive hypothesis and a single application of the congruence of $\times$ inference rule. The other cases of (7.16) are simpler, since the desired derivation sequence is one of those of the inductive hypothesis.

The induction step in the case the final equation is an instance of any of the other inference rules is carried out similarly. $\square$

It is straightforward to check that if σ conforms to the no-unit grammar, then $\mathrm{nf}_{\mathbf{T}}(\sigma) = \sigma$. We therefore have:

Corollary 7.15 *Suppose that both τ and τ' conform to the no-unit grammar. Then,*

$$\textsf{First} \vdash \tau = \tau' \Leftrightarrow \textsf{First}^- \vdash \tau = \tau'.$$

Much in the same fashion we can show

Corollary 7.16 *Suppose that both τ and τ' conform to the no-unit grammar. Then,*

$$\begin{aligned} \textsf{Linear} \vdash \tau = \tau' &\Leftrightarrow \textsf{Linear}^- \vdash \tau = \tau', \\ \textsf{Product} \vdash \tau = \tau' &\Leftrightarrow \textsf{Product}^- \vdash \tau = \tau'. \end{aligned}$$

7.4 An Algorithm for the Linear Isomorphism Problem

In this section we show a linear time and space reduction of linear isomorphism to product isomorphism. The inputs are two types τ and τ' conforming to the no-unit grammar. The algorithm outputs are two types σ, σ' such that

$$\textsf{Linear}^- \vdash \tau = \tau' \Leftrightarrow \textsf{Product}^- \vdash \sigma = \sigma'.$$

Noting that $\textsf{Linear}^-$ adds to $\textsf{Product}^-$ the currying axiom ($\mathcal{A}.6$), the algorithm converts the inputs τ and τ' into a normal form in which all curried functions are brought into an equivalent un-curried representation. This is achieved by recursively applying the anti-currying rule $\mathcal{R}.6$ to τ and τ'. The result then conforms to the un-curried grammar, in which the pattern $A \to (B \to C)$ is not allowed.

Un-curried Grammar

$$\tau ::= x \quad | \quad \tau \to x \quad | \quad \tau \to (\tau \times \tau) \quad | \quad \tau \times \tau$$

Algorithmically, the normal form is computed using function $\mathrm{nf_c}$.

$$\mathrm{nf_c}(\tau) = \begin{cases} x & \text{if } \tau = x \\ \mathrm{nf_c}(\tau_a) \times \mathrm{nf_c}(\tau_b) & \text{if } \tau = \tau_a \times \tau_b \\ R_6(\mathrm{nf_c}(\tau_a), \mathrm{nf_c}(\tau_b)) & \text{if } \tau = \tau_a \to \tau_b \end{cases} \tag{7.17}$$

If a node represents a function-type, then function R_6 checks whether the return type of this function is another function type, and if so, applies the anti-currying rule.

$$R_6(\sigma_a, \sigma_b) = \begin{cases} (\sigma_a \times \sigma_1) \to \sigma_2 & \text{if } \sigma_b = \sigma_1 \to \sigma_2 \qquad \textit{// apply rule } \mathcal{R}.6 \\ \sigma_a \to \sigma_b & \text{otherwise} \end{cases} \tag{7.18}$$

Lemma 7.17 *Let τ be a type conforming to the no-unit grammar, and let $\sigma = \mathrm{nf_c}(\tau)$. Then,* **(i)** *the call $\mathrm{nf_c}(\tau)$ executes in $O(|\tau|)$ time;* **(ii)** $\textsf{Linear}^- \vdash \tau = \sigma$*;* **(iii)** $|\sigma| = |\tau|$*; and* **(iv)** *type σ conforms to the un-curried grammar.*

PROOF. Parts **(i)**, **(ii)**, and **(iii)** are proved by structural induction, following the outline of the proof of Lemma 7.10.

In proving **(iv)** we note that there are two restrictions in the un-curried grammar. The first is that there are no occurrences of $\mathbf{T}$. This follows from the assumption that τ conforms to the no-unit grammar.

The second restriction is that the return type of all function-types is not a function-type. We show that $\mathrm{nf_c}(\tau)$ conforms to this restriction by induction on the depth of the recursive calls of $\mathrm{nf_c}$. The inductive base is the first case of (7.17) and is trivial. In the inductive step we must show that the return type of a function cannot be a function itself. A node corresponding to a function-type can be generated by $\mathrm{nf_c}$ only in the third case of (7.17). This node itself is generated by the invocation $R_6(\sigma_a, \sigma_b)$. Examining (7.18) we see that the return type of this node is σ_b precisely when σ_b is not a function-type. If however σ_b is a function-type, i.e., $\sigma_b = \sigma_1 \to \sigma_2$, then recall that σ_b was computed by a recursive application of $\mathrm{nf_c}$. Therefore, by the inductive hypothesis, σ_2, the return type of the current node is not a function-type. □

It follows from Lemma 7.17**(ii)** that if the normal forms $\mathrm{nf_c}(\tau)$ and $\mathrm{nf_c}(\tau')$ are isomorphic by applications of the commutative and associative axioms, then τ and τ' are also isomorphic by application of the commutative, associative and currying axioms, i.e.,

$$\mathsf{Product}^- \vdash \mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau') \Rightarrow \mathsf{Linear}^- \vdash \tau = \tau'. \tag{7.19}$$

The remainder of this section is dedicated to proving the converse, i.e., that after the types where brought to their un-curried normal form, all that is required in deciding isomorphism is to apply the commutative and associative axioms. The proof is similar in spirit to that of Andreev and Soloviev [6].

Lemma 7.18 $\mathsf{Linear}^- \vdash \tau = \tau' \Rightarrow \mathsf{Product}^- \vdash \mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau')$.

PROOF. The proof is by induction on the length of the derivation sequence of $\mathsf{Linear}^- \vdash \tau = \tau'$, and follows the same outline as the proof of Lemma 7.12.

The induction base is that $\tau = \tau'$ is an instance of an axiom $\mathcal{A}.0, \ldots, \mathcal{A}.6$. This cannot be one of the unit axioms $\mathcal{A}.1, \ldots, \mathcal{A}.3$ since by assumption $\mathbf{T}$ does not occur in the input. In the case that the reflexive axiom ($\mathcal{A}.0$) was applied, it is trivial to see that $\mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau')$.

In the case that this axiom was the commutative axiom ($\mathcal{A}.4$), then $\tau = \tau_a \times \tau_b$ and $\tau' = \tau_b \times \tau_a$. It is easy to see that $\mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau_a) \times \mathrm{nf_c}(\tau_b)$ and $\mathrm{nf_c}(\tau') = \mathrm{nf_c}(\tau_b) \times \mathrm{nf_c}(\tau_a)$. Therefore, $\mathsf{Product}^- \vdash \mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau')$. Similar consideration apply when this axiom was the associative axiom ($\mathcal{A}.5$).

The last axiom to consider is the currying axiom $\mathcal{A}.6$. In this case $\tau = (\tau_a \times \tau_b) \to \rho$ and $\tau' = \tau_a \to (\tau_b \to \rho)$. There are two cases to consider:

1. *Type ρ is not a function-type.* Examining the definitions (7.17) and (7.18), we find that

$$\mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau') = \Big[\mathrm{nf_c}(\tau_a) \times \mathrm{nf_c}(\tau_b)\Big] \to \mathrm{nf_c}(\rho).$$

2. *Type ρ is a function-type.* In this case we find the maximal k such that ρ can be written as

   $$\rho = \rho_1 \to \big(\rho_2 \to \cdots (\rho_{k-1} \to \rho_k) \cdots \big).$$

   Note that, by definition, ρ_k is not a function-type. Let

   $$\varrho = \mathrm{nf_c}(\rho_1) \times \Big(\mathrm{nf_c}(\rho_2) \times \cdots \times \big(\mathrm{nf_c}(\rho_{k-2}) \times \mathrm{nf_c}(\rho_{k-1}) \big) \cdots \Big).$$

   It is then easy to check that

   $$\begin{aligned} \mathrm{nf_c}(\tau) &= \Big[\big(\mathrm{nf_c}(\tau_a) \times \mathrm{nf_c}(\tau_b) \big) \times \varrho \Big] \to \mathrm{nf_c}(\rho_k), \\ \mathrm{nf_c}(\tau') &= \Big[\mathrm{nf_c}(\tau_a) \times \big(\mathrm{nf_c}(\tau_b) \times \varrho \big) \Big] \to \mathrm{nf_c}(\rho_k). \end{aligned}$$

In both cases we have that Product$^-$ $\vdash \mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau')$.

To prove the induction step we examine the last step of the derivation sequence. If this step is an axiom instance, then the same considerations as in the induction base apply. Otherwise one of the following inference rules was applied: symmetry, transitivity, congruence of $\times$, or congruence of $\to$. The only difficulty is with the congruence rule of $\to$. Consider an instance of this inference rule:

$$\frac{\tau_a = \tau_b, \tau_c = \tau_d}{\tau_a \to \tau_c = \tau_b \to \tau_d}.$$

By the inductive hypothesis, we have that Product$^-$ $\vdash \mathrm{nf_c}(\tau_a) = \mathrm{nf_c}(\tau_b)$ and Product$^-$ $\vdash \mathrm{nf_c}(\tau_c) = \mathrm{nf_c}(\tau_d)$. We would like to prove that Product$^-$ $\vdash \mathrm{nf_c}(\tau_a \to \tau_c) = \mathrm{nf_c}(\tau_b \to \tau_d)$.

Note that since Product$^-$ $\vdash \mathrm{nf_c}(\tau_c) = \mathrm{nf_c}(\tau_d)$, their root nodes have the same type, i.e., both $\mathrm{nf_c}(\tau_c)$ and $\mathrm{nf_c}(\tau_d)$ are product-types, function-types, or primitive-types. There are two cases to consider:

1. *Types $\mathrm{nf_c}(\tau_c)$ and $\mathrm{nf_c}(\tau_d)$ are both not function-types.* We find that

   $$\begin{aligned} \mathrm{nf_c}(\tau_a \to \tau_c) &= \mathrm{nf_c}(\tau_a) \to \mathrm{nf_c}(\tau_c), \\ \mathrm{nf_c}(\tau_b \to \tau_d) &= \mathrm{nf_c}(\tau_b) \to \mathrm{nf_c}(\tau_d). \end{aligned}$$

2. *Types $\mathrm{nf_c}(\tau_c)$ and $\mathrm{nf_c}(\tau_d)$ are both function-types.* Let $\mathrm{nf_c}(\tau_c) = \varrho \to \rho$ and $\mathrm{nf_c}(\tau_d) = \varrho' \to \rho'$. Since Product$^-$ $\vdash \mathrm{nf_c}(\tau_c) = \mathrm{nf_c}(\tau_d)$ we have that Product$^-$ $\vdash \varrho = \varrho'$ and Product$^-$ $\vdash \rho = \rho'$. It is then easy to check that

   $$\begin{aligned} \mathrm{nf_c}(\tau_a \to \tau_c) &= \Big[\mathrm{nf_c}(\tau_a) \times \varrho \Big] \to \mathrm{nf_c}(\rho), \\ \mathrm{nf_c}(\tau_b \to \tau_d) &= \Big[\mathrm{nf_c}(\tau_b) \times \varrho' \Big] \to \mathrm{nf_c}(\rho'). \end{aligned}$$

In both cases we have that Product$^-$ $\vdash \mathrm{nf_c}(\tau_a \to \tau_c) = \mathrm{nf_c}(\tau_b \to \tau_d)$. $\square$

7.5 Multi-set Partitioning Algorithms

For the purpose of processing product-nodes in which the terms are unsorted, we need a linear time procedure for comparing multi-sets. More generally, we develop in this section an algorithm for partitioning a collection of multi-sets of integers into equivalence classes. This algorithm runs in $O(n)$ time, where n is the size of the input representation, while using temporary (uninitialized) storage whose size is the maximal input value. Cai and Paige [21] review other linear-time algorithms for partitioning multi-sets.

Definition 7.19 (Compact integer partitioning)
Given integers $a_1, \ldots, a_n$, where $a_i \in [1, n]$ for $i = 1, \ldots, n$, the compact integer partitioning problem is to partition the input into its equivalence classes, i.e., all equal integers will be in the same partition (and only them).

The output partitioning is presented with respect to the input: Each equivalence class is produced as a list of indices, $i_1, \ldots, i_m$, such that $a_{i_1} = \cdots = a_{i_m}$. The partitioning into equivalence classes is thus represented as a list of lists of indices.

Lemma 7.20 *Compact integer partitioning can be solved in $O(n)$ time and $O(n)$ space.*

PROOF. A standard bucket sort algorithm using n buckets achieves these bounds. □

More general than compact integer partitioning is the case that the input range is not restricted to the range $[1, n]$.

Definition 7.21 (Broad integer partitioning)
Given integers $a_1, \ldots, a_n$, where $a_i \in [1, U]$ for $i = 1, \ldots, n$, the broad integer partitioning problem is to partition the input into its equivalence classes.

To deal with this problem, we first reduce the input range.

Definition 7.22 (Renaming) *Let $\mathcal{U}$ be an arbitrary domain and let $\Gamma \subseteq \mathcal{U}$, $|\Gamma| = n$. Then a partial function $\Omega : \mathcal{U} \mapsto [1, n]$ is a* renaming *of Γ if Ω is defined on Γ and for any $a, b \in \Gamma$,*

$$a \neq b \Rightarrow \Omega(a) \neq \Omega(b).$$

Algorithm 7.2 finds a renaming function for a sequence of integers drawn from the range $[1, U]$. The algorithm uses the standard trick of inverse pointers to maintain $O(1)$ access time into a sparse uninitialized array of arbitrary size. Note that main loop invariant: *After processing index i, then $\Omega[a_i] = t$ and $\mho[t] = a_i$, for some $t \in [1, \ell]$.*

Renaming makes it possible to generalize Lemma 7.20.

Lemma 7.23 *Broad integer partitioning can be solved in $O(n)$ time and $O(U + n)$ space.*

Algorithm 7.2 `Rename`$(a_1, \dots, a_n)$

Given the sequence $a_1, \dots, a_n$, where $a_i \in [1, U]$, $i = 1, \dots, n$, return **(i)** $\ell = |\{a_1, \dots, a_n\}|$ *and* **(ii)** *a renaming function represented as an array $\Omega[1, \dots, U]$, such that $\Omega[a_i]$ is a unique integer in the range $[1, \ell]$. The values of the other entries of Ω are arbitrary.*

```
1:  $\Omega \leftarrow$ new int[U] // An uninitialized array of size $U$
2:  $\mho \leftarrow$ new int[n] // The inverse mapping of $\Omega$
3:  $\ell \leftarrow 0$ // $\ell$ is the current number of distinct values in the input
4:  For $i = 1, \dots, n$ do // Compute $\Omega[a_i]$
5:      $t \leftarrow \Omega[a_i]$ // $t$ may be arbitrary if the value of $a_i$ is new
6:      If $1 \le t \le \ell$ andalso $\mho[t] = a_i$ then
7:          next $i$ // No new mapping since $a_i = a_j$ for some $j < i$
8:      else // Create a new mapping entry
9:          $\ell \leftarrow \ell + 1$ // A new distinct input value
10:         $\Omega[a_i] \leftarrow \ell$ // Store the mapping entry
11:         $\mho[\ell] \leftarrow a_i$ // Record the inverse pointer
12:     fi
13: od
```

PROOF. After applying Algorithm 7.2, we apply a *renaming process*, i.e., the replacement $a_i \leftarrow \Omega(a_i)$ for $i = 1, \dots, n$. The problem is then reduced to compact integer partitioning. □

A more general partitioning problem is when the input consists of ordered pairs.

Definition 7.24 (Pair partitioning) *Given a collection Γ of n pairs of integers*

$$\langle a_1, b_1 \rangle, \dots, \langle a_n, b_n \rangle,$$

where $a_i, b_i \in [1, U]$ for $i = 1, \dots, n$, the pair partitioning problem is to partition Γ into its equivalence classes.

Lemma 7.25 *The pair partitioning problem can be solved in $O(n)$ time and $O(U + n)$ space.*

PROOF. Apply broad integer partitioning first on $a_1, \dots, a_n$ to obtain an initial partitioning of Γ. Each of the resulting equivalence classes is then refined by broad integer partitioning with respect to the b_i's. □

Renaming with pair partitioning is also easy. Each pair is replaced by the index of its equivalence class. In fact, every partitioning algorithm gives rise to a corresponding renaming.

Lemma 7.25 can be generalized further.

Lemma 7.26 (Tuple partitioning) *Given a collection Γ of n tuples of k integers each, where each integer is drawn from the range $[1, U]$, it is possible to partition Γ into its equivalence classes, in $O(nk)$ time and $O(U + n)$ extra space.*

PROOF. Similar to Lemma 7.25, however, instead of two passes we now have k passes. The input to the first pass is the entire collection Γ, and the output is a partitioning of Γ according to the first element of each tuple.

The output of pass i is a partitioning of Γ satisfying the following invariant: *all elements in the same partition have an equal i-prefix, i.e., the same first i integers in their tuples.* Pass i refines each partition by applying broad integer partitioning according to the i^{th} element of each tuple. Since broad integer partitioning is performed in linear time, the running time of a pass is linear in the sum of partition sizes, which is exactly $n = |\Gamma|$. Thus the total running time is $O(nk)$.

At the end of the k^{th} pass the tuple partitioning problem is solved. Broad integer partitioning requires (reusable) $O(U + n)$ space. In addition, only $O(n)$ space is required for storing the current partitioning of Γ in the form of indices to the input array. $\square$

Notice that the time requirement in the above is linear in the size of the input, not the number of tuples. Also, observe that the algorithm for the tuple partitioning problem is in fact *incremental* in the sense that in the i^{th} pass we only examine the i^{th} integer in each tuple.

Corollary 7.27 (Incremental tuple partitioning)
Let Γ be a collection of n tuples of k integers each, where each integer is drawn from the range $[1, U]$. Then, it is possible to incrementally partition Γ in k passes where the i^{th} component of each tuple is specified in the i^{th} pass, in $O(n)$ time for each pass and $O(U + n)$ extra space.

A more challenging situation occurs in the case that the input consists of unordered tuples, rather than tuples. Next we will show that multi-set partitioning can also be solved in time linear in the size of the input.

Definition 7.28 (Multi-set partitioning) *Given a collection Γ of multi-sets of integers drawn from the range $[1, U]$, the multi-set partitioning problem is to partition Γ into its equivalence classes.*

Lemma 7.29 *Multi-set partitioning can be solved in $O(n)$ time and $O(U + n)$ space, where n is the sum of sizes of all multi-sets.*

PROOF. First, Algorithm 7.2 is invoked to rename all integers in the input to fit the range $[1, n]$. The next step is to sort the multi-sets. However, if each of these is sorted independently the running time would not be linear. Instead, we concatenate the sets together, prefixing each input integer with the identifier of its multi-set. All the multi-sets can then be sorted by a single application of a radix sort.

We stress that we sort the *renamed* integers, not the initial multi-sets. This process is known as *weak sort* [109]. Weak sort is possible in linear time since the renaming process is not order preserving.

Next, the ordered multi-sets are partitioned according to size. Each such partition is a collection of ordered multi-sets of equal size; in other words, each partition is a collection of tuples of equal size. All that is left is to solve the tuple partitioning problem, employing Lemma 7.26 in each partition. □

7.6 An Algorithm for the Product Isomorphism Problem

After units are eliminated, product isomorphism theory has only the commutative and associative axioms. These axioms allow products to be reordered until the two types match. Thus product isomorphism is in essence a series of multi-set partitioning problems. In this section we use the algorithms described in the previous section for these problems in developing an $O(n)$ time and space algorithm for product isomorphism. This algorithm receives two types, τ and τ', conforming to the no-unit grammar, and determines whether $\mathsf{Product}^{-} \vdash \tau = \tau'$.

The algorithm begins by *flattening all products* in the input, so that it conforms to the following product grammar.

Product Grammar

$$\rho ::= \prod_{i=1}^{k} \sigma \qquad\qquad (k \geq 1)$$
$$\sigma ::= x \quad | \quad \rho \to \rho$$

Note that we have extended the $\prod$ convention (7.2) to include products with a single term. Thus, in this grammar

$$\prod(x) = x. \tag{7.20}$$

Recall that by assumption the input cannot be isomorphic to $\mathbf{T}$, hence the start symbol ρ denotes products of at least one term. Each of these terms is either a primitive-type or a function-type.

Consider, for example, the following type, which will serve as a running example,

$$\big((\mathsf{a} \times \mathsf{b}) \to \mathsf{c}\big) \to \Big(\big(\mathsf{d} \times (\mathsf{e} \times \mathsf{f})\big) \times \big(\mathsf{g} \to (\mathsf{h} \times \mathsf{i})\big)\Big). \tag{7.21}$$

Figure 7.1 shows the expression tree of this type before and after flattening.

Algorithmically, the flattening process is carried out by computing the normal form defined by the function $\mathrm{nf_a}$. This function receives a type τ conforming to the no-unit grammar, and exhaustively applies the associative rule $\mathcal{R}.5$. The output is a type conforming to the product grammar.

$$\mathrm{nf_a}(\tau) = \begin{cases} \prod(x) & \text{if } \tau = x \\ \prod(\mathrm{nf_a}(\tau_a) \to \mathrm{nf_a}(\tau_b)) & \text{if } \tau = \tau_a \to \tau_b \\ \mathrm{nf_a}(\tau_a) \bowtie \mathrm{nf_a}(\tau_b) & \text{if } \tau = \tau_a \times \tau_b \qquad \textit{// apply rule } \mathcal{R}.5 \end{cases} \tag{7.22}$$

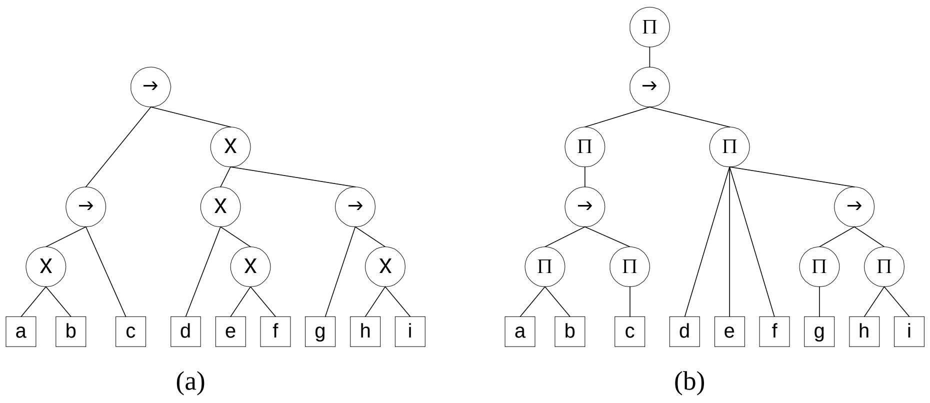


Figure 7.1: An abstract syntax tree of type (7.21) before (a) and after (b) flattening

The operation $\bowtie$ denotes the concatenation of the terms of two products, i.e.,

$$\prod_{i=1}^{k'} \tau_i \bowtie \prod_{i=k'+1}^{k} \tau_i = \prod_{i=1}^{k} \tau_i.$$

Lemma 7.30 *Let τ be a type conforming to the no-unit grammar, and let $\sigma = \mathrm{nf_a}(\tau)$. Then,* **(i)** *the call $\mathrm{nf_a}(\tau)$ executes in $O(|\tau|)$ time;* **(ii)** *$|\sigma| \leq 2|\tau|$;* **(iii)** *type σ conforms to the products grammar; and* **(iv)** Product$^-$ $\vdash \tau = \sigma$

PROOF. Trivial by structural induction. Part **(iv)** is proved by interpreting $\prod$ nodes with conventions (7.2) and (7.20) and noting that only the associative rule $\mathcal{R}.5$ was applied in the definition of $\mathrm{nf_a}(\tau)$. $\square$

The flattened representation makes it easier to decide product isomorphism. The following lemma shows how this decision might be carried out.

Lemma 7.31 *Let τ and τ' be two types conforming to the product grammar. Then,* Product$^-$ $\vdash \tau = \tau'$ *if and only if one of the following three statements holds:*

1. *Types τ and τ' are equal to the same primitive-type x.*

2. *Types τ and τ' are function-types, i.e., $\tau = \rho_1 \to \rho_2$ and $\tau' = \rho_1' \to \rho_2'$, and* Product$^-$ $\vdash \rho_1 = \rho_1'$ *and* Product$^-$ $\vdash \rho_2 = \rho_2'$.

3. *Types τ and τ' are product-types with the same number of terms, i.e., $\tau = \prod_{i=1}^{k} \sigma_i$ and $\tau' = \prod_{i=1}^{k} \sigma_i'$, and there exists a bijection $\pi : [1,k] \mapsto [1,k]$, such that* Product$^-$ $\vdash \sigma_i = \sigma'_{\pi(i)}$ *for all i, $1 \leq i \leq k$.*

PROOF. Direction $\Leftarrow$ is trivial. Direction $\Rightarrow$ is done by induction on the length of the derivation sequence of Product$^-$ $\vdash \tau = \tau'$. $\square$

The product grammar produces abstract syntax trees in which function- and product-types occur alternately on the path from the root to any leaf. We can thus define a height for each tree node, so that product (function) types are always represented by nodes of odd (even) height.

Definition 7.32 (Height) *Let τ be a type conforming to the product grammar. Then, the height of a type, denoted $h(\tau)$, is the length of the longest path from τ to any leaf, i.e.,*

$$h(\tau) = \begin{cases} 0 & \textit{if } \tau = x \\ 1 + \max_{i=1}^{k} h(\sigma_i) & \textit{if } \tau = \prod_{i=1}^{k} \sigma_i \\ 1 + \max_{i=1}^{2} h(\rho_i) & \textit{if } \tau = \rho_1 \to \rho_2 \end{cases} \tag{7.23}$$

Edges in Figure 7.1b were stretched so that nodes of the same height are drawn at the same level. Observe that product-types always have odd heights and function-types always have even heights. This can be easily proved by induction on the product grammar.

Lemma 7.33 *Let τ, τ' be two types conforming to the product grammar. Then,*

$$\textsf{Product}^{-} \vdash \tau = \tau' \Rightarrow h(\tau) = h(\tau').$$

PROOF. Trivial by structural induction on τ and τ' using Lemma 7.31. □

Theorem 7.34 *Product isomorphism can be decided in $O(n)$ time and space.*

PROOF. Consider the types represented by all of the nodes of the tree representations of τ and τ'. We will label each of these n types with an identifier drawn from the range $[1, n]$, such that two types are isomorphic if and only if they have the same identifier.

Since two types cannot be equivalent unless their heights are the same, identifiers may be assigned in ascending order of heights. Let T_ι be the set of all types of height ι. The set T_0 is the set of primitive-types. The algorithm starts by passing T_0 to the broad integer partitioning algorithm. A renaming process then yields unique identifiers for all primitive-types.

The processing of T_ι, $\iota \geq 1$ depends on whether ι is even or odd. If ι is even, then types in T_ι correspond to σ symbols in the grammar of the normal form, i.e., function-types. Equivalence among these are discovered using pair partitioning algorithm.

If however ι is odd, then the types in T_ι are products, i.e., ρ symbols. We apply the multi-set partitioning algorithm to find all equivalence classes among these.

In both even and odd levels, we apply a renaming process that assigns identifiers to types in the current level, starting at the first unused identifier.

Each node is passed to a partitioning algorithm at most twice, first in the partitioning of nodes in its height, and then as component of its parent. Therefore the total input size in all invocations of partitioning algorithms is linear, and hence the total runtime of our algorithm is linear. □

The above algorithm is applicable also in the case that types use a DAG rather than a tree representation. The runtime in this case is linear in the number of nodes *plus* the number of *edges* of the graph.

7.7 The P/F-graph

To generalize the linear isomorphism algorithm to deal with the first order isomorphism problem, we now introduce the normal form $\mathrm{nf_d}$ in which the *distributive rule* $\mathcal{R}.7$ is not applicable. As noted in Section 7.2, an exhaustive application of this rule may lead to a representation of exponential size. The $\mathbf{P}/\mathbf{F}$-graph, described in this section, is a linear size representation of the normal form $\mathrm{nf_d}$.

Let τ and τ' be two arbitrary types conforming to the product grammar. The problem is to determine whether $\mathsf{First}^- \vdash \tau = \tau'$. (The assumption that the inputs conform to the product grammar is safe since the normalizing function $\mathrm{nf_a}$ can be applied in linear time to flatten all products.)

Repeated applications of rules $\mathcal{R}.6$ and $\mathcal{R}.7$ will bring each of the inputs to the normal form defined by the *first order grammar*:

First order Grammar

$$\varrho ::= \prod_{i=1}^{k} \varsigma \qquad\qquad (k \geq 1)$$

$$\varsigma ::= x \quad | \quad \varrho \to x$$

Comparing the first order grammar and the product grammar we see that the derivation $\sigma ::= \rho \to \rho$ is replaced by $\varsigma ::= \varrho \to x$, i.e., all functions must return a primitive-type.

Algorithmically, this normal form can be generated by applying the normalizing function $\mathrm{nf_d}$, defined by

$$\mathrm{nf_d}(\tau) = \begin{cases} \prod(x) & \text{if } \tau = x \\ \bowtie_{i=1}^{k} \mathrm{nf_d}(\sigma_i) & \text{if } \tau = \prod_{i=1}^{k} \sigma_i \\ R_{6,7}\big(\mathrm{nf_d}(\rho_1), \rho_2\big) & \text{if } \tau = \rho_1 \to \rho_2 \end{cases} \tag{7.24}$$

where $R_{6,7}$ is an auxiliary function, mutually recursive with $\mathrm{nf_d}$, which handles function types:

$$R_{6,7}(\varrho, \tau) = \begin{cases} \prod(\varrho \to x) & \text{if } \tau = x \\ \bowtie_{i=1}^{k} R_{6,7}\big(\varrho, \sigma_i\big) & \text{if } \tau = \prod_{i=1}^{k} \sigma_i \qquad \textit{// apply rule } \mathcal{R}.7 \\ R_{6,7}\Big(\big(\varrho \bowtie \mathrm{nf_d}(\rho_1)\big), \rho_2\Big) & \text{if } \tau = \rho_1 \to \rho_2 \qquad \textit{// apply rule } \mathcal{R}.6 \end{cases} \tag{7.25}$$

Functions $\mathrm{nf_d}$ and $R_{6,7}$ must *eagerly* evaluate their arguments to ensure that the distributive rule is applied in outer-first order (Remark 7.9). In other words, given a function type $\rho_1 \to \rho_2$, rule $\mathcal{R}.7$ is first applied to ρ_1 and only then to ρ_2. This is the reason that the call to $R_{6,7}$ in (7.24) cannot commence before $\mathrm{nf_d}(\rho_1)$ finishes.

We shall see that the definition of $R_{6,7}$ gives rise to a multiple-terms version of the distributive transformation (7.8). In this version, an input node $\varrho \to \prod_{i=1}^{k} \sigma_i$ is converted to $\prod_{i=1}^{k}(\alpha \to \sigma_i)$ where α is represented as a pointer to the node corresponding to the product ϱ.

We now examine definitions (7.24) and (7.25) more formally. First, we show that the value returned by these functions is isomorphic to their input. Let ϱ be an arbitrary type.

Lemma 7.35

$$\begin{aligned}&\textsf{First} \vdash \tau = \mathrm{nf_d}(\tau),\\ &\textsf{First} \vdash \varrho \to \tau = R_{6,7}(\varrho, \tau).\end{aligned}$$

PROOF. We first note that since τ conforms to the product grammar, then exactly one of the three cases in the definition of either $\mathrm{nf_d}$ (7.24) or $R_{6,7}$ (7.25) must apply. The lemma is then proved by *simultaneous* structural induction on τ. The induction base is the first case in both definitions. By examining the second and third cases of (7.24) we see that it immediately follows from the (simultaneous) inductive hypothesis that function $\mathrm{nf_d}$ returns a type isomorphic to τ. The distributive (currying) axiom and the same inductive hypothesis show that $R_{6,7}$ returns a type isomorphic to $\varrho \to \tau$ in the second (third) case of its definition (7.25). □

Lemma 7.36 *Type* $\mathrm{nf_d}(\tau)$ *conforms to the first order grammar. Further, if* ϱ *also conforms to this grammar, then so does* $R_{6,7}(\varrho, \tau)$.

PROOF. Note that all types conforming to this grammar are products whose terms are either primitive or function types. The proof is again carried out by simultaneous induction on the structure of τ. Again, the induction base is trivially given by the first case of (7.24) and (7.25). The induction step is also easy: in the second case of both definition the returned value is simply a product of terms covered by the inductive hypothesis. In the third case of these definitions the returned value is of a recursive call $R_{6,7}(\cdot, \rho_2)$ where $|\rho_2| < |\tau|$. The proof is completed by checking that the first argument in both of these recursive calls conforms to the first order grammar as required for satisfying the inductive hypothesis. □

We stress that $\mathrm{nf_d}(\tau)$ may be of size $O(n^2)$, as indeed happens in example (7.10). The reason for this blowup is in the third case of $R_{6,7}$: the concatenation $\varrho \bowtie \mathrm{nf_d}(\rho_1)$ creates a new product node whose list of terms are the concatenation of two lists of terms: that of ϱ and $\mathrm{nf_d}(\rho_1)$. Note that the terms themselves are not duplicated, but a new list of terms must be created. The reason that we cannot reuse the two existing lists of terms is that ϱ can be shared among independent recursive calls due to the second case of $R_{6,7}$: we have k independent calls of the form $R_{6,7}\big(\varrho, \sigma_i\big)$.

In order to give the linear space and time bounds for the normalization process, we describe a *shared* representation of types in the first order grammar. Instead of the usual expression tree, we shall use a special rooted acyclic graph. We use the term $\mathbf{P}/\mathbf{F}$-graph since the nodes in it are either $\mathbf{P}$-nodes (representing product-types) or $\mathbf{F}$-nodes (representing function-types).

A $\mathbf{P}$-node v has a field $\varphi(v)$ storing the non-empty set of pointers to term nodes. Terms are either $\mathbf{F}$-nodes or primitive-types, which are encoded simply by identifiers in the range $[1, n]$. In addition, v has a field $\texttt{parent}(v)$ pointing to another $\mathbf{P}$-node, from which v inherits additional terms.

An $\mathbf{F}$-node u has a field $\texttt{arg}(u)$, which is a pointer to the $\mathbf{P}$-node storing the function argument type, and a field $\texttt{ret}(u)$, which is a primitive-type specifying the function return type.

$\mathbf{P}/\mathbf{F}$-graphs are further restricted by the demand that `parent` edges define a tree over the $\mathbf{P}$-nodes called the *product tree*, and denoted $\mathcal{T}$. The tree $\mathcal{T}$ is rooted at a dummy $\mathbf{P}$-node, denoted $\mathbf{P}_\perp$, which has no terms, i.e., $\varphi(\mathbf{P}_\perp) = \emptyset$. $\mathbf{P}$-nodes are therefore initialized with their `parent` field pointing at $\mathbf{P}_\perp$.

Definition 7.37 (Expanded terms) *The expanded terms of a* $\mathbf{P}$-node v*, denoted* $\phi(v)$*, are the union of terms of its ancestors in the product tree, i.e.,* $\phi(v) = \varphi(v) \cup \phi(\texttt{parent}(v))$*, where* $\phi(\mathbf{P}_\perp) = \emptyset$*.*

Consider, for example, Figure 7.2a which shows type (7.21) in the product grammar. Figure 7.2b shows the result of applying algorithm *`NormalizeProduct`* (described later) on this type.

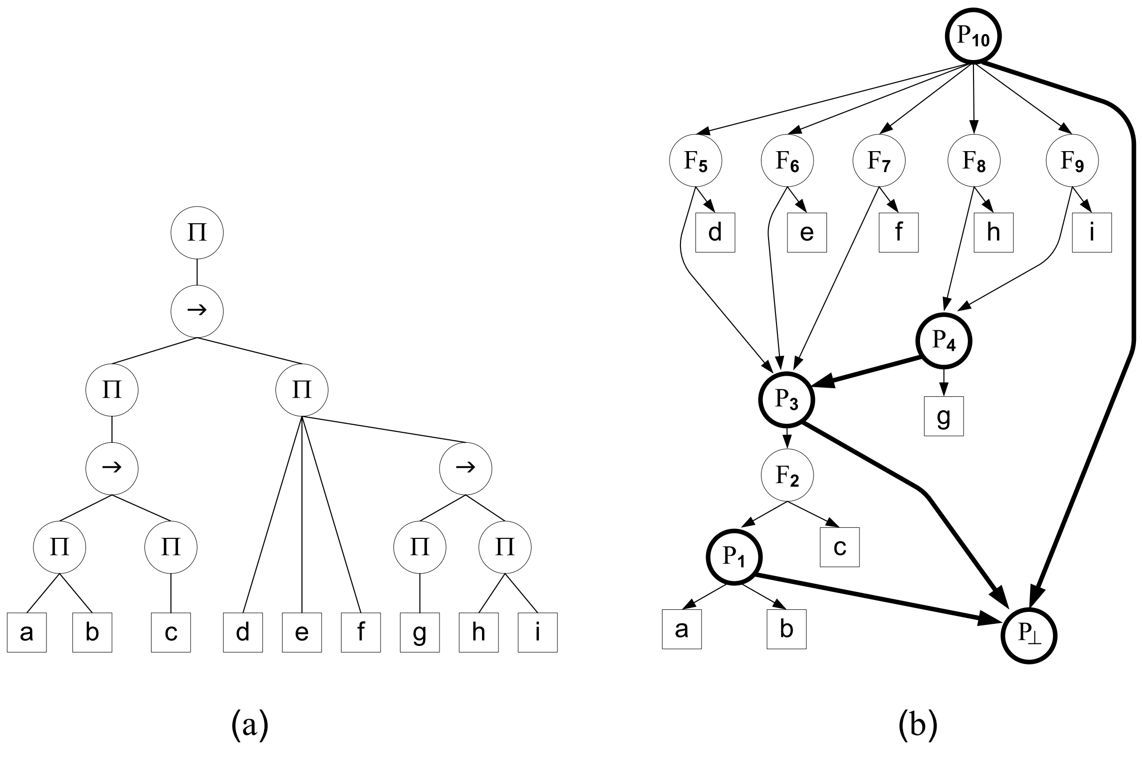


Figure 7.2: (a) Type (7.21) in the product grammar, and (b) its $\mathbf{P}/\mathbf{F}$-graph representation. The `parent` edges are depicted in bold.

The $\mathbf{P}$-nodes in Figure 7.2b are:

$$\begin{array}{rcll} \mathbf{P}_\perp = & & & \prod \\ \mathbf{P}_1 = & \mathbf{P}_\perp & \bowtie & \prod(\mathsf{a}, \mathsf{b}) \\ \mathbf{P}_3 = & \mathbf{P}_\perp & \bowtie & \prod(\mathbf{F}_2) \\ \mathbf{P}_4 = & \mathbf{P}_3 & \bowtie & \prod(\mathsf{g}) \\ \mathbf{P}_{10} = & \mathbf{P}_\perp & \bowtie & \prod(\mathbf{F}_5, \mathbf{F}_6, \mathbf{F}_7, \mathbf{F}_8, \mathbf{F}_9) \end{array} \tag{7.26}$$

We see that each term of a $\mathbf{P}$-node is either a primitive type (e.g., a) or an $\mathbf{F}$-node (e.g., $\mathbf{F}_2$). In addition to the set of terms, each $\mathbf{P}$-node (except $\mathbf{P}_\perp$) inherits additional terms via the `parent` edge. For example, $\texttt{parent}(\mathbf{P}_4) = \mathbf{P}_3$, i.e., $\mathbf{P}_4$ inherits the terms of $\mathbf{P}_3$ which recursively inherits the terms of $\mathbf{P}_\perp$. Therefore, the extended terms of $\mathbf{P}_4$ are the union of the terms of $\mathbf{P}_4$, $\mathbf{P}_3$, and $\mathbf{P}_\perp$:

$$\phi(\mathbf{P}_4) = \varphi(\mathbf{P}_4) \cup \varphi(\mathbf{P}_3) \cup \varphi(\mathbf{P}_\perp).$$

Algorithm 7.3 and Algorithm 7.4 present two mutually recursive routines, namely *`NormalizeProduct`* and *`FunctionIntoProduct`*, respectively. These routines are storage-minded variants of functions nf_d and $R_{6,7}$, respectively. Together, the two describe a single pass traversal of an abstract syntax tree of a type conforming to the product grammar. The output is a linear sized $\mathbf{P}/\mathbf{F}$-graph of an isomorphic type in the first order grammar.

Algorithm 7.3 *`NormalizeProduct`*(τ)

Given a type τ conforming to the product grammar, return a $\mathbf{P}$-node v *of an isomorphic type in the first order grammar.*

```
1:  $v \leftarrow$ new $\mathbf{P}$-node // Initially $\texttt{parent}(v) = \mathbf{P}_\perp$, $\varphi(v) = \emptyset$
2:  If $\tau$ is a primitive-type $x$ then
3:      $\varphi(v) \leftarrow \{x\}$
4:  else if $\tau$ is a product-type then
5:      Let $k$ and $\sigma_i$, $i = 1, \ldots, k$, be such that $\tau = \prod_{i=1}^{k} \sigma_i$
6:      For $i = 1, \ldots, k$ do // Normalize all terms in the product
7:          $u_i \leftarrow \texttt{NormalizeProduct}(\sigma_i)$
8:          $\varphi(v) \leftarrow \varphi(v) \cup \varphi(u_i)$ // Collect terms of $u_i$
9:      od
10: else // $\tau$ is a function-type
11:     Let $\rho_1$ and $\rho_2$ be such that $\tau = \rho_1 \rightarrow \rho_2$
12:     $u \leftarrow \texttt{NormalizeProduct}(\rho_1)$
13:     $v \leftarrow \texttt{FunctionIntoProduct}(u, \rho_2)$
14: fi
15: Return $v$
```

Lines 2–3 of Algorithm 7.3 correspond to the first case of function nf_d, lines 4–9 to the second case, and lines 10–14 to the third. The union operation in line 8 correspond to the concatenation operation $\bowtie$ in the second case of nf_d.

Algorithm 7.4 follows the same outline as function $R_{6,7}$: lines 2–5 correspond to the first case of $R_{6,7}$, lines 6–11 to the second, and lines 12–17 to the third. Again, the

Algorithm 7.4 `FunctionIntoProduct`(u, τ)

Given a $\mathbf{P}$-node u *and a type* τ *(which is a product-type) return* v*, a new* $\mathbf{P}$-node *describing a type isomorphic to the function-type* $\varrho \to \tau$*, where* ϱ *is the type represented by the* $\mathbf{P}$-node u*.*

```
 1: $v \leftarrow$ new $\mathbf{P}$-node // Initially parent$(v) = \mathbf{P}_\perp$, $\varphi(v) = \emptyset$
 2: If $\tau$ is a primitive-type $x$ then
 3:     $w \leftarrow$ new $\mathbf{F}$-node
 4:     arg$(w) \leftarrow u$; ret$(w) \leftarrow x$ // $w$ represents the type $\varrho \to x$
 5:     $\varphi(v) \leftarrow \{w\}$
 6: else if $\tau$ is a product-type then
 7:     Let $k$ and $\sigma_i$, $i = 1, \ldots, k$, be such that $\tau = \prod_{i=1}^{k} \sigma_i$
 8:     For $i = 1, \ldots, k$ do // Normalize all terms in the product
 9:         $u_i \leftarrow$ FunctionIntoProduct$(u, \sigma_i)$
10:         $\varphi(v) \leftarrow \varphi(v) \cup \varphi(u_i)$ // Collect terms of $u_i$
11:     od
12: else // $\tau$ is a function-type
13:     Let $\rho_1$ and $\rho_2$ be such that $\tau = \rho_1 \to \rho_2$
14:     $w \leftarrow$ NormalizeProduct$(\rho_1)$
15:     parent$(w) \leftarrow u$ // Share the common argument $\varrho$
16:     $v \leftarrow$ FunctionIntoProduct$(w, \rho_2)$
17: fi
18: Return $v$
```

union operation in line 10 correspond to the concatenation operation $\bowtie$ in the second case of $R_{6,7}$. However, the concatenation operation $\bowtie$ in the third case of $R_{6,7}$ was translated into an assignment to the `parent` field of w in line 15. This line is the crux of the two routines, making the linear space representation possible.

Let us examine lines 12–17 and the third case of $R_{6,7}$. Node u represents type ϱ, and node w represents the product $\varrho \bowtie \mathrm{nf_d}(\rho_1)$. In line 14, we assign `NormalizeProduct`(ρ_1) to w. Then, instead of adding the terms of u to w (i.e., $\varphi(w) \leftarrow \varphi(w) \cup \varphi(u)$) we point the `parent` field of w to u in line 15. Therefore the expanded terms of w are equal to those of the product $\varrho \bowtie \mathrm{nf_d}(\rho_1)$.

The next lemma proves that algorithms 7.3 and 7.4 run in $O(n)$ time and space.

Lemma 7.38 *Let* τ *be a type conforming to the product grammar, and let* u *be a* $\mathbf{P}$-node*. Then, the function calls* `NormalizeProduct`(τ) *and* `NormalizeProduct`(u, τ) *execute in* $O(|\tau|)$ *time and space.*

PROOF. Proved by mutually-recursive structural-induction on τ. The induction base is when τ is a primitive type. It is mundane to check that lines 2–3 of Algorithm 7.3 and lines 2–5 of Algorithm 7.4 execute in constant time and space. In the induction step, τ is either a function or a product. The amount of time and space invested in addition to the recursive calls is either constant if $\tau = \rho_1 \to \rho_2$ or $O(k)$ if $\tau = \prod_{i=1}^{k} \sigma_i$. Note that the union in line 8 of Algorithm 7.3 and line 10 of Algorithm 7.4 can be computed in constant time since the terms of u_i are not shared (in contrast to the terms of u which are shared among other calls). $\square$

The following lemma shows that first order isomorphism of two types can be decided by bringing each of these types into their $\mathbf{P}/\mathbf{F}$ representation, and then traversing the two graphs in tandem, comparing at each stage the expanded terms of the current nodes.

Lemma 7.39 *Two nodes u, v in a $\mathbf{P}/\mathbf{F}$-graph represent isomorphic types if and only if one of the following three statements holds:*

1. *Nodes u and v represent the same primitive-type x.*
2. *Nodes u and v are both* $\mathbf{F}$-nodes*, $\texttt{ret}(u) = \texttt{ret}(v)$ and $\texttt{arg}(u)$ and $\texttt{arg}(v)$ (recursively) represent isomorphic types.*
3. *Nodes u and v are both* $\mathbf{P}$-nodes*, and there exists a bijection π from $\phi(u)$ to $\phi(v)$, such that every $v' \in \phi(u)$ (recursively) represents a type isomorphic to $\pi(v')$.*

PROOF. Let τ and τ' be the types u and v represent, respectively. Then, both τ and τ' conform to the first order grammar. Rittri [116] proved that, in such a case (i.e., when none of the rules $\mathcal{R}.1$–$\mathcal{R}.7$ can be applied), we have that

$$\mathsf{First} \vdash \tau = \tau' \Leftrightarrow \mathsf{Product}^{-} \vdash \tau = \tau'.$$

Deciding the latter can be done using Lemma 7.31. $\square$

If the terms in $\mathbf{P}$-nodes are expanded, then the size of the representation may increase to $O(n^2)$ (as in (7.10)). With this expansion, the problem becomes an instance of product isomorphisms, which, as explained in the previous section, can be solved in linear time. We can thus obtain a simple $O(n^2)$ time and space algorithm for the first order isomorphism problem, thereby improving upon the $O(n^2 \log n)$ best previous result. To obtain a more efficient algorithm, we develop in the next two sections the machinery for comparing unexpanded products.

7.8 Tree Partitioning

We need to further develop our partitioning algorithms to deal with the *non-expanded* representation of products in the tree of $\mathbf{P}$-nodes rooted at $\mathbf{P}_\perp$. The partitioning of these nodes is tantamount to finding the type isomorphism relationships between $\mathbf{P}$-nodes: Two $\mathbf{P}$-nodes are in the same equivalence class of the partitioning when the *expanded* terms of the respective nodes are the same, which happens if and only if the types these two nodes represent are isomorphic.

To understand this need better, consider again our running example type (7.21)

$$\big((\mathsf{a} \times \mathsf{b}) \rightarrow \mathsf{c}\big) \rightarrow \Big(\big(\mathsf{d} \times (\mathsf{e} \times \mathsf{f})\big) \times \big(\mathsf{g} \rightarrow (\mathsf{h} \times \mathsf{i})\big)\Big).$$

Algorithm `NormalizeProduct` generated the $\mathbf{P}/\mathbf{F}$-graph representation of this type. This representation is depicted again in Figure 7.3a below.

By definition, removing all $\mathbf{F}$-nodes and the edges incident on them from a $\mathbf{P}/\mathbf{F}$-graph will result in a tree. Figure 7.3b shows the tree thus obtained from Figure 7.3a. As explained above, the extended terms of each $\mathbf{P}$-node are computed by inheriting the extended terms of its parent (see Definition 7.37). For example, tree node $\mathbf{P}_4$ in the figure inherits the terms of tree node $\mathbf{P}_3$.

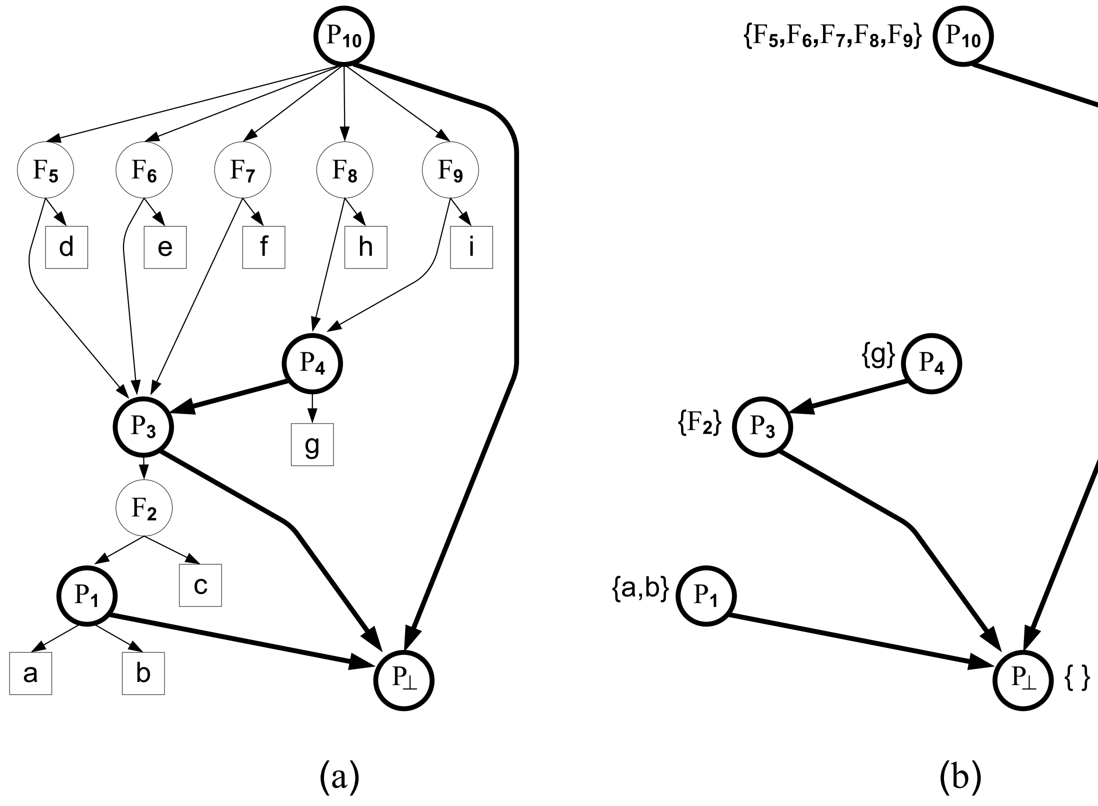


Figure 7.3: (a) $\mathbf{P}/\mathbf{F}$-graph representation in Figure 7.2b, and (b) its product-tree with the multi-set of terms of each product.

Let us ignore the $\mathbf{F}$-nodes for now, and concentrate on a variant of the multi-set partitioning problem in which the multi-sets are defined by an inheritance tree. We will first develop an algorithm for this variant. Still, we note that this algorithm does not completely solve the general problem of sorting the nodes of a $\mathbf{P}/\mathbf{F}$-graph into equivalence classes. The reason is that the terms in the product-tree are not always known in advance. In Figure 7.3b we see for example that the term $\mathbf{F}_6$ in $\mathbf{P}_{10}$ is not available upfront. We need to process node $\mathbf{P}_3$ before we can be certain that this term is not isomorphic to, for example, term $\mathbf{F}_8$, which in turn depend upon $\mathbf{P}_4$. The next section will take care of this subtlety by developing an incremental algorithm for the problem.

In this section, our concern lies with the simpler, non-incremental, setting, described as follows: Given is a tree $\mathcal{T}$ of n nodes such that a multi-set $\varphi(v)$ of integers is associated with each node $v \in \mathcal{T}$. The *expanded multi-set* of a node v is the union of multi-sets of the ancestors of v, i.e.,

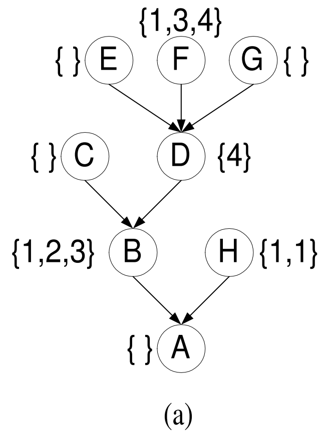
$$\phi(v) = \bigcup_{u \preceq v} \varphi(u).$$

These expanded multi-sets will be in our applications the expanded terms (Definition 7.37) of $\mathbf{P}$-nodes.

Definition 7.40 (Tree partitioning) *Given a tree $\mathcal{T}$, the* tree partitioning *is the partitioning defined by the multi-set partitioning of the expanded multi-sets $\{\phi(v) \mid v \in \mathcal{T}\}$.*

Let M denote the total number of elements in multi-sets of $\mathcal{T}$, i.e., $M = \sum_{v \in \mathcal{T}} |\varphi(v)|$. We can assume that the integers in the input to the problem are condensed so that $\bigcup_{v \in \mathcal{T}} \varphi(v) = [1, m]$. (This condition can be ensured by a simple application of a renaming process.)

Figure 7.4a shows an example of a tree with $n = 8$ nodes with their associated multi-sets (only four of which are non-empty). In the example, $m = 4$ distinct integers take part in these multi-sets. The total number of elements in these multi-sets is $M = 9$.



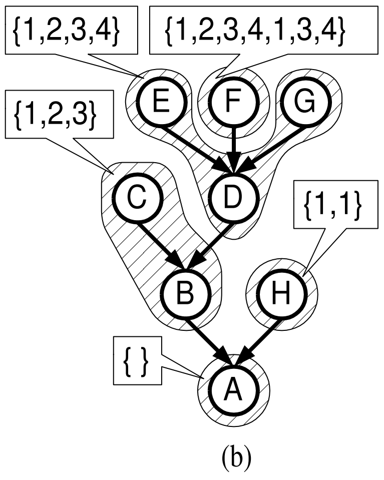


Figure 7.4: A small multi-set tree (a) and its tree partitioning (b)

We have for nodes E and F, for instance,

$$\varphi(\mathsf{E}) = \emptyset$$
$$\varphi(\mathsf{F}) = \{1, 3, 4\}$$
$$\phi(\mathsf{E}) = \{1, 2, 3, 4\}$$
$$\phi(\mathsf{F}) = \{1, 2, 3, 4, 1, 3, 4\}$$

Figure 7.4b depicts the solution of the tree partitioning problem for the multi-set tree of Figure 7.4a. We see that there are 5 partitions:

$$\{\mathsf{A}\}, \{\mathsf{H}\}, \{\mathsf{B}, \mathsf{C}\}, \{\mathsf{D}, \mathsf{E}, \mathsf{G}\}, \{\mathsf{F}\}. \tag{7.27}$$

The callout attached to each partition shows the expanded multi-set of all nodes in this partition. For example, $\{1, 2, 3, 4\}$ is the expanded multi-set of the partition $\{\mathsf{D}, \mathsf{E}, \mathsf{G}\}$.

The *naïve solution* to the tree partitioning problem is by directly computing the expanded multi-sets $\phi(v)$. In order to do so, we represent an expanded multi-set $\phi(v)$ as an integer array $\mathbf{Count}_v[1,\ldots,m]$.

Definition 7.41 *Given an expanded multi-set $\phi(v)$, its* array-representation, *denoted $\mathbf{Count}_v$, is an array over the indices $[1,\ldots,m]$, such that $\mathbf{Count}_v[i]=k$ if integer i occurs k times in $\phi(v)$.*

Array $\mathbf{Count}_v$ can be easily computed from $\varphi(v)$ and $\mathbf{Count}_u$, where u is v's parent. After having obtained the arrays $\mathbf{Count}_v$, the tree partitioning problem becomes the partitioning problem of these arrays, viewed as m-sized tuples. The total size of those n arrays is nm cells, while the time required for computing them is $O(nm+M)$ time since we also examined all the terms $\varphi(v)$. To conclude, the runtime of the naïve solution is $O(nm+M)$ while using $O(nm)$ space.

We now present an algorithm for finding the tree partitioning whose total runtime is $O(M\log m)$ using $O(M)$ space. This algorithm relies on the *dual* representation in which, instead of associating a multi-set of integers with each node, a multi-set of nodes is associated with each integer. (To simplify the complexity analysis we assume that $n\le M$. This assumption is true in our application since $\mathbf{P}$-nodes have a non-empty set of terms, i.e., $|\varphi(u)|\ge 1$.)

Definition 7.42 *A* family *F_i, $i=1,\ldots,m$, is a multi-set of nodes such that if i occurs k times in $\varphi(v)$, then v occurs k times in F_i.*

In our example, four such families are defined:

$$\begin{aligned}F_1&=\{\mathsf{B},\mathsf{F},\mathsf{H},\mathsf{H}\},\\F_2&=\{\mathsf{B}\},\\F_3&=\{\mathsf{B},\mathsf{F}\},\\F_4&=\{\mathsf{D},\mathsf{F}\}.\end{aligned}\tag{7.28}$$

Note that $\sum_{i=1}^{m}|F_i|=M$.

Given a tree $\mathcal{T}$ and a multi-set F of its nodes, it is easy to define a partitioning of the nodes of $\mathcal{T}$ where the classification criterion is the number of occurrences of a node in F. We shall however be interested in a more sophisticated such partitioning, denoted ∇F, in which the classification criterion is the number of times a node “inherits” membership in F. More precisely,

Definition 7.43 *Let u,v be two nodes of $\mathcal{T}$, and let $\mathrm{ancestors}(u)$ (respectively, $\mathrm{ancestors}(v)$) be the set of ancestors of u. Then, u and v are in the same partition of ∇F if and only if*

$$|\mathrm{ancestors}(u)\cap F|=|\mathrm{ancestors}(v)\cap F|.$$

In our example, the four family partitionings induced by the families of (7.28) are:

$$\begin{aligned}\nabla F_1 &= \{\{\mathsf{A}\},\{\mathsf{F},\mathsf{H}\},\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{G}\}\},\\ \nabla F_2 &= \{\{\mathsf{A},\mathsf{H}\},\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{F},\mathsf{G}\}\},\\ \nabla F_3 &= \{\{\mathsf{A},\mathsf{H}\},\{\mathsf{F}\},\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{G}\}\},\\ \nabla F_4 &= \{\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{H}\},\{\mathsf{F}\},\{\mathsf{D},\mathsf{E},\mathsf{G}\}\}.\end{aligned} \tag{7.29}$$

Note that all the nodes in a certain partition of ∇F_i, $1 \le i \le 4$, have the same number of occurrences of i. For example, $\mathbf{Count}_{\mathsf{F}}[1] = \mathbf{Count}_{\mathsf{H}}[1] = 2$. In fact, it is easy to prove the following:

Lemma 7.44 *Let F_i be a family, and v be a node of $\mathcal{T}$, then*

$$|\text{ancestors}(v) \cap F_i| = \mathbf{Count}_v[i].$$

The performance gain of the dual representation is due to the fact that the multi-set of nodes in which a value participates is often a subtree of $\mathcal{T}$. For example, the partition $\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{F},\mathsf{G}\}$ of ∇F_2 is a subtree rooted at B.

Next we define the *intersection* of two partitionings P_1 and P_2, written as $P_1 \times P_2$, and show that $\nabla F_1 \times \cdots \times \nabla F_m$ is in fact the tree partitioning.

Definition 7.45 *Let P_1 and P_2 be two partitionings. Then, their* intersection*, denoted $P_1 \times P_2$, is defined by*

$$P_1 \times P_2 = \{p_1 \cap p_2 \mid p_1 \in P_1, p_2 \in P_2\}.$$

In other words $P_1 \times P_2$ is obtained by intersecting each partition of P_1 with each partition of P_2. For example, the intersection of ∇F_1 and ∇F_2 is

$$\begin{aligned}\nabla F_1 \times \nabla F_2 &= \{\{\mathsf{A}\},\{\mathsf{F},\mathsf{H}\},\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{G}\}\} \times \{\{\mathsf{A},\mathsf{H}\},\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{F},\mathsf{G}\}\}\\ &= \{\{\mathsf{A}\},\{\mathsf{H}\},\{\mathsf{F}\},\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{G}\}\}.\end{aligned}$$

It is mundane to see that $\times$ is commutative and associative.

Lemma 7.46 *The partitioning $\nabla F_1 \times \cdots \times \nabla F_m$ is the tree partitioning.*

PROOF. Let P be the tree partitioning. Let $u, v \in \mathcal{T}$ be arbitrary. For a partitioning X, we write $u \equiv v \bmod X$ to denote that u, v belong to the same partition of X. Then we need to prove that $u \equiv v \bmod P$ if and only if

$$u \equiv v \bmod \nabla F_1 \times \cdots \times \nabla F_m.$$

Suppose first that

$$u \equiv v \bmod P. \tag{7.30}$$

Then, from the definition of the tree partitioning (Definition 7.40) we have that

$$\phi(u) = \phi(v). \tag{7.31}$$

It follows by the definition of the array-representation $\mathbf{Count}[1, \ldots, m]$ (Definition 7.41) that

$$\forall 1 \le i \le m \bullet \mathbf{Count}_u[i] = \mathbf{Count}_v[i]. \tag{7.32}$$

If $\mathbf{Count}_u[i] = \mathbf{Count}_v[i]$ then, by Lemma 7.44, $|\mathrm{ancestors}(u) \cap F_i| = |\mathrm{ancestors}(v) \cap F_i|$, so we may write

$$\forall 1 \le i \le m \bullet |\mathrm{ancestors}(u) \cap F_i| = |\mathrm{ancestors}(v) \cap F_i|. \tag{7.33}$$

From the definition of the ∇ operator (Definition 7.43) we have that

$$\forall 1 \le i \le m \bullet u \equiv v \bmod \nabla F_i. \tag{7.34}$$

Finally, from the definition of the intersection of two partitionings (Definition 7.45)

$$u \equiv v \bmod \nabla F_1 \times \cdots \times \nabla F_m. \tag{7.35}$$

To show that (7.30) follows from (7.35) we trivially follow the above reasoning chain in the reverse direction. $\square$

We now devise an efficient representation of family partitionings and a way to compute their intersection. To this end, we describe below the *segmented-array* representation of a family partitioning ∇F which requires $O(|F|)$ space. We also show how to intersect two segmented-arrays A_1 and A_2, which results in another segmented-array A_3 which represents $A_1 \times A_2$ where

$$|A_3| \le |A_1| + |A_2|.$$

The trick is to consider a pre-order traversal of the tree, in which subtrees can be simply encoded as intervals. Therefore, members of a family F define intervals, which in turn break the pre-order into segments. Thus, the partitioning ∇F can be encoded as an array mapping those segments to their containing partition.

In our example, let the pre-order traversal be

$$\pi = (\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{H}).$$

As can be seen in Figure 7.5, the descendants of any given node form an interval. This figure highlights the intervals of the descendants of nodes B and F:

$$\begin{aligned} &\mathrm{descendants}(\mathsf{B}) = \{\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}\} = [\mathsf{B}, \mathsf{G}], \\ &\mathrm{descendants}(\mathsf{F}) = \{\mathsf{F}\} = [\mathsf{F}, \mathsf{F}]. \end{aligned}$$

Consider now the family F_3 defined by these two nodes, $F_3 = \{\mathsf{B}, \mathsf{F}\}$. In Figure 7.5 we see that the two intervals of F_3,

$$\mathrm{Intervals}(F_3) = \{[\mathsf{B}, \mathsf{G}], [\mathsf{F}, \mathsf{F}]\},$$

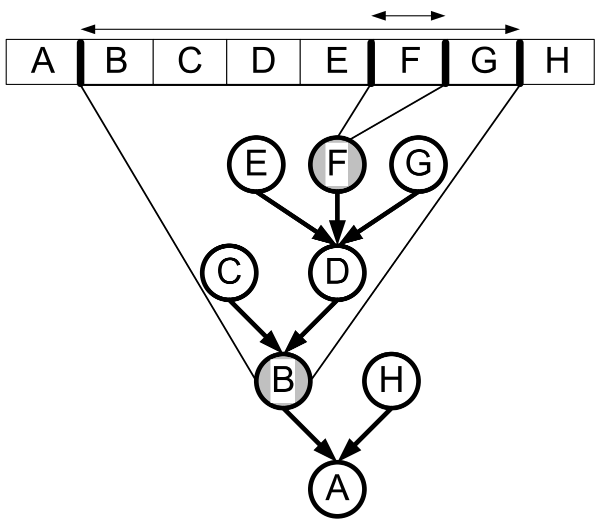


Figure 7.5: The intervals and segments defined by family $F_3 = \{\mathsf{B}, \mathsf{F}\}$

break π into five segments

$$\text{Segments}(F_3) = \{[\mathsf{A}, \mathsf{A}], [\mathsf{B}, \mathsf{E}], [\mathsf{F}, \mathsf{F}], [\mathsf{G}, \mathsf{G}], [\mathsf{H}, \mathsf{H}]\}.$$

Consider any arbitrary such segment defined by F_3, and let v range over the nodes of this segment. Then, the multiplicity of the value 3 in $\phi(v)$ is the same, e.g., the multiplicity of the value 3 in the segment $[\mathsf{B}, \mathsf{E}]$ is 1. The *segmented-array* representation associates a multiplicity to each segment. This multiplicity is called the *segment descriptor*. The segmented-array of family F_3 is therefore

$$\text{SegmentedArray}(F_3) = \langle [\mathsf{A}, \mathsf{A}] \mapsto 0, [\mathsf{B}, \mathsf{E}] \mapsto 1, [\mathsf{F}, \mathsf{F}] \mapsto 2, [\mathsf{G}, \mathsf{G}] \mapsto 1, [\mathsf{H}, \mathsf{H}] \mapsto 0\rangle,$$

and its family partitioning is

$$\nabla F_3 = \{\{\mathsf{A}, \mathsf{H}\}, \{\mathsf{F}\}, \{\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{G}\}\}.$$

Observe that each segment is contained in some partition of ∇F_3, and that two segments with the same descriptor belong to the same partition. For example, both segments $[\mathsf{B}, \mathsf{E}]$ and $[\mathsf{G}, \mathsf{G}]$ are contained in the partition $\{\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{G}\}$ of ∇F_i. In fact, the union of those two segments is exactly this partition. It is easy to check that this is no coincidence, i.e., the union of segments with the same descriptor is equal to some partition in ∇F_3, and vice versa.

More formally,

Definition 7.47 *Let P be a partitioning of the nodes of $\mathcal{T}$, and let π be a pre-order traversal of $\mathcal{T}$. Then, a* segmented-array *representation of P is an array of segment records, each record containing the segment starting and ending indices and a descriptor such that:*

1. *The segments are distinct and cover π, i.e., the segments are a partitioning of π.*

2. *Each segment is contained in some partition of P. In other words, the segmented array represents a finer-grained partitioning than P.*

3. *Two segments have the same descriptor if and only if they are contained in the same partition of P.*

4. *The segments are sorted in an increasing order.*

We will sometimes refer to a family partitioning ∇F as a segmented-array. No confusion will arise.

A segmented-array representation of a family partitioning ∇F can be created in $O(|F|)$ time and space since the number of segments is linear in $|F|$. More precisely, a family F defines at most $|F|$ distinct intervals in π, one for each distinct node in F. These intervals break π into at most $2|F| + 1$ segments.

Figure 7.6 depicts the segmented-array representations of the family partitionings of (7.29).

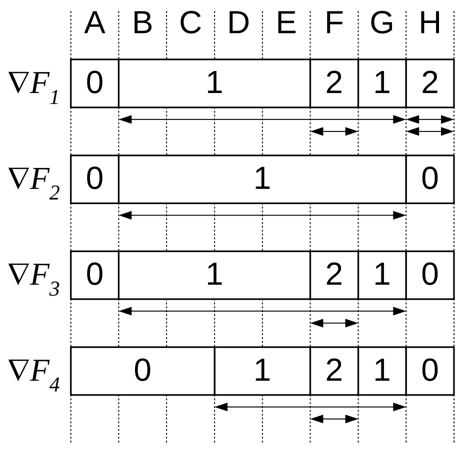


Figure 7.6: The segmented-arrays of the families of Figure 7.4a

The *intersection* of two segmented-arrays P_1 and P_2, whose sizes are s_1 and s_2, is carried out by merging their arrays in $O(s_1+s_2)$ time into a single array of size at most s_1+s_2. The descriptors of the segments in $P_1 \times P_2$ are the *renamed* pairs of descriptors of the originating segments from P_1 and P_2 (using Lemma 7.25).

Figure 7.7 depicts the intersection of the segmented-arrays of ∇F_1 and ∇F_2 from (7.28).

The third row in the figure shows the intermediate stage in which the segments in the intersection still use pairs of integers as descriptors. For example, $\langle 1, 1\rangle$ is the descriptor of the segment containing nodes B, C, D, and E. This descriptor was renamed to 1. Note that the other segment (singleton with G) with the pair descriptor $\langle 1, 1\rangle$ was also renamed to 1.

We are now ready to state the principal result of this section describing the (non-incremental) tree partitioning algorithm and its performance.

Theorem 7.48 *There is an $O(M \log m)$ time and $O(M)$ space algorithm solving the tree partitioning problem.*

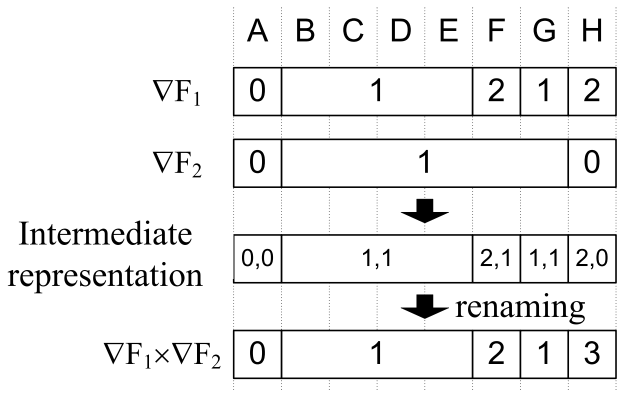


Figure 7.7: Computing the intersection of the two segmented-arrays ∇F_1 and ∇F_2 defined by Figure 7.4a.

PROOF. Using Lemma 7.46, we wish to compute $\nabla F_1 \times \cdots \times \nabla F_m$. We therefore build a balanced binary tree whose leaves are the segmented-arrays $\nabla F_1, \ldots, \nabla F_m$. In each internal node we compute the intersection of the two segmented-arrays of its two children. The segmented-array at the root of this tree represents the tree partitioning.

Consider the first level of this tree which contains the segmented-arrays $\nabla F_1, \ldots, \nabla F_m$. Recall that the size of the segmented-array ∇F_i is $2|F_i| + 1$. Therefore, the size of the entire first level is

$$\sum_{i=1}^{m}(2|F_i| + 1) = O(M).$$

In calculating the second level of the tree, we intersect pairs of segmented-arrays $\nabla F_i \times \nabla F_{i+1}$, for odd values of i. Recall also that the time (and space) for creating $\nabla F_i \times \nabla F_{i+1}$ is $O(|F_i|+|F_{i+1}|)$. Thus, the time (and space) for creating the second level is again $O(M)$.

In general, since all the segmented-arrays propagate to the root, we have that the total size of all segmented-arrays at each tree level, and thus the work to generate the next level, is $O(M)$. Since the number of levels is $\lceil \log_2 m \rceil + 1$, we have that the total time for computing $\nabla F_1 \times \cdots \times \nabla F_m$ is $O(M \log m)$. $\square$

For an example, refer to Figure 7.8 which depicts the balanced binary tree of the families of (7.28). We see in the figure that the segmented-array at the root of this binary tree, i.e., $\nabla F_1 \times \nabla F_2 \times \nabla F_3 \times \nabla F_4$, partitions the ordering π into 6 segments. The segment of types D and E has $\mathrm{id} = 2$. This is also the id of the segment of G. Together, these two segments represent the partition $\{\mathsf{D}, \mathsf{E}, \mathsf{G}\}$. We have thus obtained the desired partitioning (7.27) of the tree in Figure 7.4a.

7.9 Incremental Tree Partitioning

The tree partitioning problem (Definition 7.40) solved in the previous section does not capture in full the intricacies of the bottom up classification into isomorphism classes of

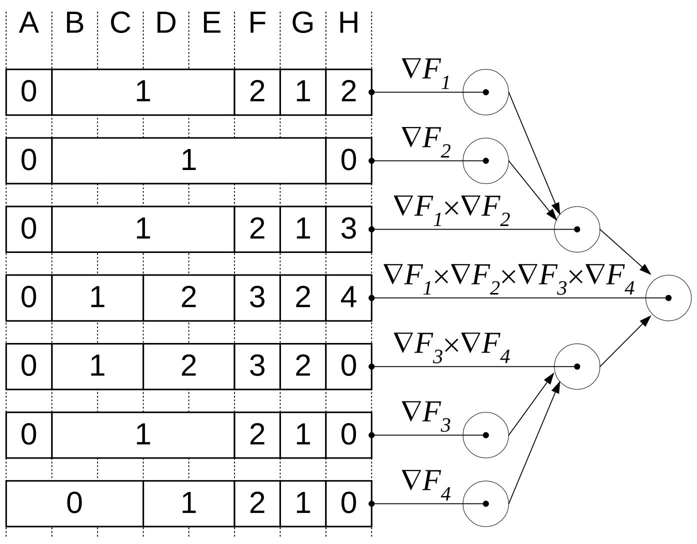

A B C D E F G H
0 1 2 1 2
0 1 0
0 1 2 1 3
0 1 2 3 2 4
0 1 2 3 2 0
0 1 2 1 0
0 1 2 1 0
∇F_1
∇F_2
$\nabla F_1 \times \nabla F_2$
$\nabla F_1 \times \nabla F_2 \times \nabla F_3 \times \nabla F_4$
$\nabla F_3 \times \nabla F_4$
∇F_3
∇F_4

Figure 7.8: The balanced binary tree of the families of Figure 7.4

the nodes of a $\mathbf{P}/\mathbf{F}$-graph. The difficulty is that the terms of $\mathbf{P}$-nodes in any given height are $\mathbf{F}$-nodes. These $\mathbf{F}$-nodes must be classified prior to the classification of the $\mathbf{P}$-nodes in this height. The algorithm behind Theorem 7.48 however assumes that all multi-sets members are directly comparable. It is applicable only in the case when all terms are primitive-types.

In this section, we develop the algorithm which after having classified all the $\mathbf{P}$-nodes up to height ι, will use this information to classify the $\mathbf{F}$-nodes in height $\iota + 1$. The identifier found in the classification of these $\mathbf{F}$-nodes must take part in the classification of the $\mathbf{P}$-nodes at height $\iota + 2$.

To this end, this section deals with a more general variant of the tree partitioning problem, in which the multi-sets are supplied in a *piecemeal fashion*. In this variant, the different possible values of the multi-sets in the tree nodes are exposed in iterations. The algorithm for this variant will add another logarithmic factor to the time complexity.

The requirements from a data structure for the *incremental tree partitioning problem* are best defined in terms of the dual representation.

Definition 7.49 *Given a tree $\mathcal{T}$, an* incremental tree partitioning data structure *must support two kinds of operations, which might be interleaved:*

1. *Operation* `insert`(F_j)*, where F_j is a family, i.e., a multi-set of nodes of $\mathcal{T}$.*

2. *Query* `classify`(T_k)*, where T_k is a subset of the nodes of $\mathcal{T}$. This query returns the tree partitioning of T_k according to the families inserted so far. More formally, let $\{F_1, \ldots, F_j\}$ be the set of families inserted so far. Then, the query returns the restriction of $\nabla F_1 \times \cdots \times \nabla F_j$ to the set T_k. This restriction is defined in the obvious manner, i.e., it is the partitioning obtained by intersecting each partition of $\nabla F_1 \times \cdots \times \nabla F_j$ with T_k, and ignoring all thusly obtained empty partitions.*

To make the complexity analysis easier, we assume that the sets $\{T_k\}$ are disjoint, that $\bigcup_k T_k = \mathcal{T}$ and that the data structure is never required to classify a node before its parent.

These assumptions hold in our application: the set of nodes T_k is exactly the set of **P**-nodes whose height is $2i$, and a family F_j is inserted after having discovered that a certain collection of **F**-nodes belong in the isomorphism class whose identifier is j. (These identifiers are allocated consecutively.)

Our main objective is to minimize the resources for processing the entire interleaved sequence of data structure operations. The next theorem states the performance characteristics of our incremental tree partitioning algorithm.

Theorem 7.50 *Incremental tree partitioning can be solved in $O(M \log m + n \log n \log m)$ time and $O(M)$ space.*

PROOF. We use a lazy representation of an infinite complete binary tree, similar to the binary tree of Theorem 7.48, The leaves of this tree are given by the infinite sequence $\nabla F_1, \nabla F_2, \ldots$

Figure 7.9 shows (part of) this tree, after families $\nabla F_1, \ldots, \nabla F_7$ have been inserted.

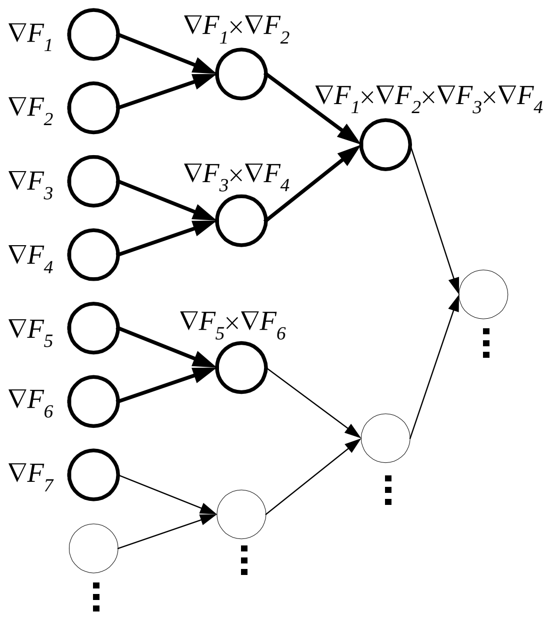

∇F_1
∇F_2
∇F_3
∇F_4
∇F_5
∇F_6
∇F_7
$\nabla F_1 \times \nabla F_2$
$\nabla F_3 \times \nabla F_4$
$\nabla F_5 \times \nabla F_6$
$\nabla F_1 \times \nabla F_2 \times \nabla F_3 \times \nabla F_4$

Figure 7.9: An embedding of seven families into an infinite balanced binary tree

This infinite tree is used to guide the computation of the intersection of the partitioning which were inserted so far: we delay the intersection of partitionings in an internal node until *both* its children exist. A *temporary root* is a node in which the partitioning was computed, but not in its parent.

In the figure the nodes at which partitionings were intersected are drawn with thicker lines. Specifically, at this stage we have computed $\nabla F_1 \times \nabla F_2$, $\nabla F_3 \times \nabla F_4$, $\nabla F_1 \times \nabla F_2 \times \nabla F_3 \times \nabla F_4$, and $\nabla F_5 \times \nabla F_6$. There are three temporary roots in figure, which are the nodes corresponding to $\nabla F_1 \times \nabla F_2 \times \nabla F_3 \times \nabla F_4$, $\nabla F_5 \times \nabla F_6$ and ∇F_7.

Assume that a new family F_8 is inserted. We first calculate its segmented-array ∇F_8, and proceed to compute the following three intersections:

$$P_1 = \nabla F_8 \times \nabla F_7,$$
$$P_2 = P_1 \times (\nabla F_5 \times \nabla F_6),$$
$$P_3 = P_2 \times (\nabla F_1 \times \nabla F_2 \times \nabla F_3 \times \nabla F_4).$$

After this insertion we will have a single temporary root.

The total time for all insert operations, i.e., $\texttt{insert}(F_1), \ldots, \texttt{insert}(F_m)$, is the same as in the non-incremental tree partitioning problem, i.e., $O(M \log m)$ time using $O(M)$ space.

The algorithm is lazy in the sense that we do not compute the intersection of the temporary roots $P_1, \ldots, P_r$. Instead, the classification of a set T_k, i.e., $\texttt{classify}(T_k)$ query, is carried out by consulting the segmented-arrays at those temporary roots. Recall that P_i is represented as a sorted array of segment-identifier pairs (see Definition 7.47). Since the size of this array is bounded by n, we can support searches in P_i in $O(\log n)$ time. For each $v \in T_k$, we search for the descriptor of the segment which contains v, in P_i for $i = 1, \ldots, r$.

After obtaining an r-tuple of descriptors for all $v \in T_k$, we apply a tuple partitioning algorithm to classify T_k. In order to keep the space linear, we cannot actually store $|T_k|$ tuples of length r. Therefore, we will use the *incremental tuple partitioning algorithm*. Specifically, we will use $|T_k|$ memory cells to find the first elements of the tuples, pass them to the tuple partitioning algorithm, and proceed to find the second elements of the tuples, etc.

Note that after j families were inserted, there are at most $\lceil \log_2 j \rceil$ temporary roots, so we always have that $r \leq \lceil \log_2 m \rceil$. Thus, the total time for computing the r-tuple is $O(r \log n) \subseteq O(\log m \log n)$. The total time for the $\texttt{classify}(T_k)$ query is therefore $O(|T_k| \log m \log n)$, while using $O(M)$ space. Since every node $v \in \mathcal{T}$ can take part in a classification query at most once, the total time for all classifications is $O(n \log n \log m)$.

The total time for all insertion operations and all classification queries is $O(M \log m + n \log n \log m)$, while the total space used is $O(M)$. $\square$

7.10 Conclusion: An Algorithm for the First Order Isomorphism Problem

Having developed the algorithms for generating the linear size $\mathbf{P}/\mathbf{F}$ representation, and for efficiently comparing the multi-sets ϕ without actually creating them, we are ready to describe the main result of this paper: an efficient algorithm for deciding first order isomorphisms. In essence, the algorithm uses Lemma 7.39. A naive recursive application of the lemma may lead to an exponential running time. To bound the time complexity, we instead traverse the graphs bottom-up, classifying the nodes into their isomorphisms equivalence classes as we do so.

The bottom-up traversal is guided by height, where all nodes of the same height are processed together. Height is defined as in Definition 7.32. Algorithm 7.5 shows how heights can be computed in linear time even in the non-expanded, $\mathbf{P}/\mathbf{F}$ representation.

Algorithm 7.5 `Height` (v)

Given a node v in a $\mathbf{P}/\mathbf{F}$-graph, ensure that $h(v')$ stores the height of v' for all nodes v' reachable from v and return $h(v)$.

```
1: If $v$ was visited then
2:     Return $h(v)$
3: fi
4: mark $v$ as visited
5: If $v$ is a primitive-type or $v = \mathbf{P}_\perp$ then
6:     $h(v) \leftarrow 0$; return $h(v)$ // Recursion base
7: fi
8: If $v$ is an $\mathbf{F}$-node then
9:     $h(v) \leftarrow 1 +$ Height (arg$(v)$); return $h(v)$
10: fi
    // $v$ must be an ordinary $\mathbf{P}$-node
11: $h(v) \leftarrow$ Height (parent$(v)$)
12: For all $u \in \varphi(v)$ do // recurse on all (non-expanded) terms
13:     $h(v) \leftarrow \max(h(v), 1 +$ Height $(u))$
14: od
15: Return $h(v)$
```

Given a node v, the algorithm uses a standard recursive depth first search to visit, compute and store the height of every node v' reachable from v. Lines 8–9 deal with the case that v is an $\mathbf{F}$-node. The recursive call in this case is only on `arg`(u), since `ret`(v) must be a primitive-type.

Another easy case is that v is $\mathbf{P}_\perp$. Since there are no terms in this product-node, its height is 0. Lines 11–15 deal with ordinary $\mathbf{P}$-nodes. The height of such nodes is one more than the maximum height of all expanded terms. The reason why in line 11 we do not add 1 to *`Height`* (`parent`(v)) is that the expanded terms include the terms $\phi($`parent`$(v))$, and not `parent`(v) as a term.

Once the height of all nodes in $\mathbf{P}/\mathbf{F}$-graph is computed, Algorithm 7.6 can be invoked to partition these nodes into equivalence classes. We assume that unique identifiers, drawn from the range $[1, n]$, are given to all primitive-types. To process non-primitive-types, the algorithm relies on the fact that nodes cannot represent isomorphic types unless they are of the same kind and the same height. Accordingly, the nodes of G are processed by height.

The main data-structure used by the algorithm is incremental tree partitioning (see Theorem 7.50). Nodes at odd height are $\mathbf{P}$-nodes. The classification of these nodes is carried out by querying this data-structure.

Lines 10–17 in the algorithm take care of $\mathbf{F}$-nodes. Classification of these nodes is carried out by a simple pair partitioning algorithm. We then generate identifiers for each of the isomorphism classes. All $\mathbf{F}$-nodes take parts as terms of $\mathbf{P}$-nodes. We must make sure that two $\mathbf{F}$-nodes in the same isomorphism class are regarded as equal when

Algorithm 7.6 `NodesPartitioning` (G)

Given a $\mathbf{P}/\mathbf{F}$*-graph* G *representing a type in the first order grammar, return a partitioning* Λ *of all the nodes of* G *into equivalence classes, such that two nodes are in the same class if and only if they represent isomorphic types.*

```
 1: Let $\Upsilon$ be an incremental tree partitioning data-structure for the tree of $\mathbf{P}$-nodes of $G$
 2: $j \leftarrow 0$ // The identifier of current isomorphism class
 3: Let $r$ be the root of $G$
 4: $l \leftarrow$ Height $(r)$
 5: For $\iota = 1, \ldots, l$ do // Process the nodes by height
 6:    Let $T_\iota \leftarrow \{v \in G \mid h(v) = \iota\}$
 7:    If $\iota$ is odd then // $T_\iota$ is a collection of $\mathbf{P}$-nodes
 8:       $\Lambda \leftarrow \Lambda \cup \Upsilon$.classify$(T_\iota)$
 9:    else // $T_\iota$ is a collection of $\mathbf{F}$-nodes
10:       Partition $T_\iota$ using pair partitioning
11:       Let the resulting partition be $T_\iota = C_1 \cup \cdots \cup C_k$
12:       $\Lambda \leftarrow \Lambda \cup \{C_1, \ldots, C_k\}$
          // Update $\Upsilon$
13:       For $i = 1, \ldots, k$ do // Inserting a new family
14:          $j \leftarrow j + 1$ // Process a new isomorphism class $j$
15:          Let $F_j$ be the multi-set of $\mathbf{P}$-nodes with a term in $C_i$
16:          $\Upsilon$.insert$(F_j)$
17:       od
18:    fi
19: od
20: Return $\Lambda$
```

comparing $\mathbf{P}$-nodes in the next iteration. Line 15 defines the multi-set F_j of $\mathbf{P}$-nodes in which isomorphic $\mathbf{F}$-nodes are terms. Note that F_j is a multi-set since a $\mathbf{P}$-node may have several terms belonging to C_i. In line 16 the incremental tree partitioning data structure is updated.

Lemma 7.51 *If* G *has* n *nodes and* $O(n)$ *edges then, Algorithm 7.6 runs in* $O(n \log^2 n)$ *time and while consuming* $O(n)$ *space.*

PROOF. We first note that computing the height as in Algorithm 7.5 requires linear time, since every node and every edge is visited at most once.

The algorithm uses linear space, since the two main procedures it invokes: incremental tree partitioning algorithm (lines 8 and 16) and pair partitioning (line 10) use linear space.

The running time of all the applications of the pair partitioning algorithm is $O(n)$ (see Lemma 7.25).

The total number of families inserted is $O(n)$. Moreover, the total size of those families is also $O(n)$, and all the sets of classified nodes are disjoint. Therefore, using Theorem 7.50, the total time of all the operations performed on Υ is

$$O(M \log m + x \log x \log m)$$

while using $O(M)$ space, where x is the number of nodes in the product-tree (which is the number of $\mathbf{P}$-nodes), m is the number of families, and M is the total size of those families. Since all the above parameters are $O(n)$, the total runtime is $O(n\log^2 n)$ using $O(n)$ space. □

The bottom-up node classification of Algorithm 7.6 can be used to solve the first order isomorphism problem. To do so, we first create the $\mathbf{P}/\mathbf{F}$-graphs of the two input types, and then merge these graphs, by e.g., making their roots descendants of a new $\mathbf{P}$-node. (The $\mathbf{P}_\perp$ nodes of the respective graphs must be unified.) Algorithm 7.6 is then invoked on the merged graph. The inputs are isomorphic if and only if these two roots are placed in the same equivalence class.

Theorem 7.52 *First order isomorphism can be decided in $O(n\log^2 n)$ time and $O(n)$ space, where n is the size of the input.*

PROOF. As noted above the $\mathbf{P}/\mathbf{F}$-graph representation uses linear space. Moreover, bringing the input to this representation requires linear time.

The complexity of comparing inputs in the $\mathbf{P}/\mathbf{F}$-graph representation is given by Lemma 7.51. □

7.11 Open Problems

The only lower bound for the first order type isomorphism problem is the trivial information theoretic linear time. An important research direction is to bridge this gap by either *reducing the time complexity* of our main algorithm even further, or obtaining better *lower bounds*.

For example, *dynamic fractional cascading* [95] might be used to decrease the running time from $O(n\log^2 n)$ to $O(n\log n\log\log n)$. Recall that in the incremental tree partitioning algorithm (Section 7.9) a `classify` query was implemented by conducting *independent* logarithmic time searches in $O(\log n)$ temporary roots. The fractional cascading data structure makes it possible to use the result of each search in expediting the subsequent search, bringing down the runtime of `classify`(T_k) to $O(|T_k|\log n\log\log n)$. Unfortunately, this representation makes it difficult to use the incremental tuple partitioning algorithm, and increases the space to $O(n\log n)$.

Time complexity might be improved also by taking the perspective in which primitive types are thought of as variables, while compound types are considered expressions over these. Then, it follows from the fact that axioms $\mathcal{A}.1$–$\mathcal{A}.7$ are complete [19] that the first order isomorphism problem is reduced to function identity. This identity might in turn be checked by an appropriate random assignment to the variables, possibly leading to a more time efficient, yet *randomized* algorithm for the problem. For example, if infinite precision arithmetic is allowed, then, it might be possible to extend the type isomorphism heuristics of Katzenelson, Pinter and Schenfeld [85], and check identity by assigning into the variables values drawn at random from, say, the range $[0,1]$. We note however that such a randomized algorithm does not yield the *isomorphism proof* as does our deterministic algorithm.

Another interesting direction comes from the generalization in which type expression trees may share nodes, i.e., the input is *directed acyclic graph* rather than a tree. This situation occurs naturally in programming languages in which non-primitive types can be named, and where these names can be used in the definition of more complex types.

Perhaps the most important problem which this chapter leaves open is efficient algorithms for *subtyping* (of products, functions, or both) which include the distributive and the currying axioms.

Chapter 8

Conclusions

The *object-oriented* (OO) paradigm, and the OO languages that enable it, such as C++ and JAVA, has become the norm for software development. In this thesis, we have developed efficient algorithms for the core features of OO languages: subtyping tests, method dispatching and object layout. **These algorithms have *the potential* of reducing the space and time overhead of any OO application.**

The efficiency of these algorithms was demonstrated on a collection of huge hierarchies drawn from as many as eight different OO languages, with the purpose of making our research *language independent*. We have used the following three *efficiency metrics*: (i) space, (ii) query time, and (iii) time required for creation the encoding. We showed significant space savings and fast creation time, usually without compromising the query time metric [136–139]. Space savings can have a crucial impact on embedded applications, and can also reduce the *total runtime* due to cache behavior and reduced page faults. We could not measure such benefits to the total runtime because we are missing two things:

1. an implementation inside a compiler for a certain language, and
2. a data-set of *applications* drawn from that specific language. The hierarchies in our data-set were drawn from various languages, and extracted from huge *production systems* rather than *applications*.

After filling these two gaps we will also be able to gather statistics on other metrics such as *code space*.

Here are three important directions for future research:

Dynamic Benchmarks The presented algorithms could be incorporated into a compiler such as Jikes RVM. Then it would be possible to gather statistics on dynamic benchmarks as opposed to our static benchmarks which are missing runtime behavior. It is also important to fine tune these theoretical algorithms to come close to the practical optimum rather than the theoretical one, using the gathered statistics.

For example, the *incremental type slicing* (TS) scheme [137] can be used for JAVA's subtyping tests, and the *incremental compact dispatch table* (CT) algorithm [138] for JAVA's `invokeinterface` instruction. It would also be interesting to consider a batch version of JAVA, i.e., where the whole program is known at compile time and dynamic loading is prohibited. Such a close world assumption is reasonable for embedded systems, and has been made in SYNERJY [1], FLEX [2], MARMOT [3], and MANTA [4]. For the batch variant, *PQ-encoding* (PQE) [136] can be used for subtyping tests, and *type slicing* (TS) [137] for dispatching. I am currently in touch with Dominique Colnet from INRIA Lorraine, for implementing the *two-dimensional bi-directional* (TDBD) object layout scheme [139] in the SMALLEIFFEL environment [135].

New algorithms The ultimate objective of practical research of the implementation of runtime environment of OO programs is a *unified object model* which would offer significant improvements over current implementations. Such a model should support, in the multiple inheritance setting, subtyping, single- and multiple-dispatching queries, updates to the hierarchy, and a good object layout. Many examples of open problems can be found in our papers [136–140], such as incremental dispatching in the multiple inheritance setting and incremental multiple dispatching even in the single inheritance setting.

New applications Partial orders are widespread throughout many disciplines. In computer science they have importance in querying data-bases (e.g., finding transitive-closure), programming languages (e.g., multiple inheritance), operating systems (e.g., virtual time in distributed systems), computational linguistics, knowledge representation, and machine learning. Our techniques for incrementally maintaining the subtyping relation can therefore have applications in any of these fields.

Surprisingly, we have discovered that our techniques for method dispatch [137,138] could be used for solving the first order isomorphism problem [140], a problem that is unrelated to partial orders or OO languages. It would be interesting to find other applications for techniques described in my thesis.

[1] a JAVA-like synchronous Language for Embedded Controllers, developed by Fraunhofer Institute for Autonomous Intelligent Systems

[2] a compiler infrastructure written in JAVA for Java, developed in MIT

[3] an optimizing compiler for JAVA, developed by Microsoft

[4] a native source-to-binary JAVA compiler, developed in Vrije, Amsterdam

Bibliography

[1] R. AGRAWAL, A. BORGIDA, AND H. V. JAGADISH, *Efficient management of transitive relationships in large data and knowledge bases*, in Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, J. Clifford, B. G. Lindsay, and D. Maier, eds., Portland, Oregon, 31 May–2 June 1989, ACM Press, pp. 253–262.

[2] R. AGRAWAL, L. DEMICHIEL, AND B. LINDSAY, *Static type checking of multi-methods*, in Proceedings of the 6*th* Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Phoenix, Arizona, USA, Oct. 6-11 1991, OOPSLA'91, ACM SIGPLAN Notices 26(11) Nov. 1991.

[3] B. ALPERN, A. COCCHI, S. FINK, D. GROVE, AND D. LIEBER, *Efficient implementation of* JAVA *interfaces:* `invokeinterface` *considered harmless*, in OOPSLA'01 [103].

[4] B. ALPERN, A. COCCHI, AND D. GROVE, *Dynamic type checking in Jalapeño*, in Java Virtual Machine Research and Technology Symposium, J. Clifford, B. G. Lindsay, and D. Maier, eds., Monterey, California, Apr. 2001, USENIX.

[5] E. AMIEL, O. GRUBER, AND E. SIMON, *Optimizing multi-method dispatch using compressed dispatch tables*, in Proceedings of the 9*th* Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon, USA, Oct. 23-27 1994, OOPSLA'94, ACM SIGPLAN Notices 29(10) Oct. 1994, pp. 244–258.

[6] A. ANDREEV AND S. SOLOVIEV, *A deciding algorithm for linear isomorphism of types with complexity* $O(n\log^2(n))$, Category Theory and Computer Science, 1290 (1997), pp. 197–209.

[7] K. ARNOLD AND J. GOSLING, *The Java Programming Language*, The Java Series, Addison-Wesley, Reading, Massachusetts, 1996.

[8] J. AUERBACH, C. BARTON, AND M. RAGHAVACHARY, *Type isomorphisms with recursive types*, Tech. Report RC 21247, IBM Research Division, Yorktown Heights, New York, August 1998.

[9] J. AUERBACH AND M. C. CHU-CARROLL, *The mockingbird system: A compiler-based approach to maximally interoperable distributed programming*, Tech. Report RC 20178, IBM Research Division, Yorktown Heights, New York, February 1997.

[10] G. BARTHE AND O. PONS, *Type isomorphisms and proof reuse in dependent type theory*, in Proc. of FOSSACS'01, vol. 2030, 2001, pp. 57–71.

[11] D. A. BASIN, *Equality of terms containing associative-commutative functions and commutative binding operators is isomorphism complete*, in 10th International Conference on Automated Deduction, Springer-Verlag New York, Inc., 1990, pp. 251–260.

[12] G. D. BATTISTA AND R. TAMASSIA, *On-line planarity testing*, Technical Report CS-89-31, Brown University - Department of Computer Science, May 1989.

[13] ——, *On-line graph algorithms with SPQR-trees*, in Automata, Languages and Programming, 17th International Colloquium, M. S. Paterson, ed., vol. 443 of Lecture Notes in Computer Science, Warwick University, England, 16–20 July 1990, Springer-Verlag, pp. 598–611.

[14] G. D. BATTISTA AND R. TAMASSIA, *On-line maintenance of triconnected components with SPQR-trees*, Algorithmica, 15 (1996), pp. 302–318.

[15] D. G. BOBROW, L. G. DEMICHIEL, R. P. GABRIEL, S. E. KEENE, G. KICZALES, AND D. A. MOON, *Common Lisp object system specification*. X3J13 Document 88-002R, June 1988.

[16] D. G. BOBROW, K. KAHN, G. KICZALES, L. MASINTER, M. STEFIK, AND F. ZDYBEL, *CommonLoops: Merging Lisp and object-oriented programming*, in OOPSLA'86 [104], pp. 17–29.

[17] K. S. BOOTH AND G. S. LEUKER, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms.*, J. Comput. Sys. Sci., 13 (1976), pp. 335–379.

[18] V. BOUCHITTE AND M. MORVAN, eds., *International Workshop on Orders, Algorithms, and Applications (ORDAL'94)*, no. 831 in Lecture Notes in Computer Science, Lyon, France, July 1994, Springer Verlag.

[19] K. B. BRUCE, R. DI COSMO, AND G. LONGO, *Provable isomorphisms of types*, Mathematical Structures in Computer Science, 1 (1991), pp. 1–20.

[20] K. B. BRUCE AND G. LONGO, *Provable isomorphisms and domain equations in models of typed languages*, in Proc. of the 7th annual ACM symposium on Theory of computing, ACM Press, May 1985, pp. 263–272.

[21] J. CAI AND R. PAIGE, *Using multiset discrimination to solve language processing problems without hashing*, Theoretical Computer Science, 145 (1995), pp. 189–228.

[22] C. CAPELLE, *Representation of an order as union of interval orders*, in Bouchitte and Morvan [18], pp. 143–162.

[23] L. CARDELLI AND P. WEGNER, *On understanding types, data abstractions, and polymorphism*, ACM Comput. Surveys, 17 (1985), pp. 471–522.

[24] T. CARGILL, B. COX, W. COOK, M. LOOMIS, AND A. SNYDER, *Is multiple inheritance essential to OOP?* Panel discussion at the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95) (Washington, DC), Oct. 1993.

[25] Y. CASEAU, *Efficient handling of multiple inheritance hierarchies*, in OOPSLA'93 [105], pp. 271–287.

[26] C. CHAMBERS, *The Cecil language, specification and rationale*, Tech. Report TR-93-03-05, University of Washington, Seattle, 1993.

[27] C. CHAMBERS AND W. CHEN, *Efficient multiple and predicate dispatching*, in OOPSLA'99 [108], pp. 238–255.

[28] W. CHEN, V. TURAU, AND W. KLAS, *Efficient dynamic look up strategy for multi-methods*, in Proceedings of the 8*th* European Conference on Object-Oriented Programming [56], pp. 408–431.

[29] N. H. COHEN, *Type-extension tests can be performed in constant time*, ACM Trans. Prog. Lang. Syst., 13 (1991), pp. 626–629.

[30] T. COHEN AND J. Y. GIL, *Self-calibration of metrics of Java methods*, in Proceedings of the 26*rd* International Conference on Technology of Object-Oriented Languages and Systems, Sydney, Australia, Nov. 20-23 2000, TOOLS Pacific 2000, Prentice-Hall, pp. 94–106.

[31] T. J. CONROY AND E. PELEGRI-LLOPART, *An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations*, Addison-Wesley, Menlo Park,CA 94025, 1983.

[32] J. CONSIDINE, *Deciding isomorphisms of simple types in polynomial time*, tech. report, CS Department, Boston University, April 2000.

[33] W. R. COOK, *A proposal for making Eiffel type-safe*, in Proceedings of the 3*rd* European Conference on Object-Oriented Programming, Nottingham, UK, July 10-14 1989, ECOOP'89, Cambridge University Press, pp. 17–29.

[34] A. CORSARO AND R. CYTRON, *Efficient memory-reference checks for real-time java*, in Proceedings of 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03), San Diego, California, USA, June11–13 2003, ACM, pp. 51–58.

[35] P. COUSOT AND R. COUSOT, *Abstract interpretation and application to logic programs*, Journal of Logic Programming, 13 (1992), pp. 103–179.

[36] B. J. COX, *Object-Oriented Programming - An Evolutionary Approach*, Addison-Wesley, Reading, Massachusetts, 1986.

[37] P. DEUTSCH AND A. SCHIFFMAN, *Efficient implementation of the Smalltalk-80 system*, in 11*th* Symposium on Principles of Programming Languages, POPL'84, Salt Lake City, Utah, Jan. 1984, ACM SIGPLAN — SIGACT, ACM Press, pp. 297–302.

[38] R. Di Cosmo, *Type isomorphisms in a type-assignment framework*, in Proc. of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, 1992, pp. 200–210.

[39] ——, *Isomorphisms of types: from λ-calculus to information retrieval and language design*, Birkhauser, 1995. ISBN-0-8176-3763-X.

[40] P. F. Dietz, *Maintaining order in a linked list*, in Proc. of the 14th Ann. ACM Symp. on Theory of Computing, San Francisco, California, United States, 1982, ACM Press, pp. 122–127.

[41] P. F. Dietz and D. D. Sleator, *Two algorithms for maintaining order in a list*, in Proc. of the 19th Ann. ACM Symp. on Theory of Computing, New York, New York, United States, 1987, ACM Press, pp. 365–372.

[42] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, *Dynamic perfect hashing: Upper and lower bounds*, SIAM J. Comput., 23 (1994), pp. 738–761.

[43] E. W. Dijkstra, *Recursive programming*, Numerische Mathematik, 2 (1960), pp. 312–318.

[44] R. Dixon, T. McKee, M. Vaughan, and P. Schweizer, *A fast method dispatcher for compiled languages with multiple inheritance*, in Proceedings of the 4th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, New Orleans, Louisiana, Oct. 1-6 1989, OOPSLA'89, ACM SIGPLAN Notices 24(10) Oct. 1989, pp. 211–214.

[45] K. Driesen, *Selector table indexing & sparse arrays*, in OOPSLA'93 [105], pp. 259–270.

[46] ——, *Software and hardware techniques for efficient polymorphic calls*, Technical Report TRCS99-24, University of California, Santa Barbara. Computer Science., July 15, 1999.

[47] K. Driesen and U. Hölzle, *Minimizing row displacement dispatch tables*, in OOPSLA'95 [106], pp. 141–155.

[48] ——, *The direct cost of virtual functions calls in C++*, in Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, San Jose, California, Oct. 6-10 1996, OOPSLA'96, ACM SIGPLAN Notices 31(10) Oct. 1996, pp. 306–323.

[49] K. Driesen, U. Hölzle, and J. Vitek, *Message dispatch on modern computer architectures*, Technical Report TRCS94-20, University of California, Santa Barbara. Computer Science., Feb. 9, 1995.

[50] ——, *Message dispatch on pipelined processors*, in Proceedings of the 9th European Conference on Object-Oriented Programming, no. 952 in Lecture Notes in Computer Science, Aarhus, Denmark, Aug. 7–11 1995, ECOOP'95, Springer Verlag, pp. 253–282.

[51] E. DUJARDIN, *Efficient dispatch of multimethods in constant time using dispatch trees*, Technical Report RR-2892, Inria, Institut National de Recherche en Informatique et en Automatique, 1996.

[52] E. DUJARDIN, E. AMIEL, AND E. SIMON, *Fast algorithms for compressed multimethod dispatch table generation*, ACM Trans. Prog. Lang. Syst., 20 (1998), pp. 116–165.

[53] N. ECKEL AND J. Y. GIL, *Empirical study of object-layout strategies and optimization techniques*, in Proceedings of the 14th European Conference on Object-Oriented Programming, no. 1850 in Lecture Notes in Computer Science, Sophia Antipolis and Cannes, France, June 12–16 2000, ECOOP 2000, Springer Verlag, pp. 394–421.

[54] ECOOP 2003, *Proceedings of the 17th European Conference on Object-Oriented Programming*, no. 2743 in Lecture Notes in Computer Science, Darmstadt, Germany, July 21–25 2003, Springer Verlag.

[55] ECOOP'91, *Proceedings of the 5th European Conference on Object-Oriented Programming*, no. 512 in Lecture Notes in Computer Science, Geneva, Switzerland, July15–19 1991, Springer Verlag.

[56] ECOOP'94, *Proceedings of the 8th European Conference on Object-Oriented Programming*, no. 821 in Lecture Notes in Computer Science, Bologna, Italy, July 4-8 1994, Springer Verlag.

[57] ELLIS AND B. STROUSTRUP, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, Massachusetts, Jan. 1994.

[58] A. FALL, *Heterogeneous encoding*, in Proceedings of International KRUSE'95 Conference: Knowledge Use, Retrieval and Storage for Efficiency, G. Ellis, R. Levinson, A. Fall, and V. Dahl, eds., Santa Cruz, California, Aug. 1995, Department of Computer Science, University of California at Santa Cruz, USA, pp. 162–167.

[59] ——, *Sparse term encoding for dynamic taxonomies*, in Proceedings of the Fourth International Conference on Conceptual Structures (ICCS-96): Knowlegde Representation as Interlingua, P. W. Eklund, G. Ellis, and G. Mann, eds., vol. 1115 of LNAI, Berlin, Aug. 19–22 1996, Springer, pp. 277–292.

[60] P. FERRAGINA AND S. MUTHUKRISHNAN, *Efficient dynamic method-lookup for object oriented languages*, in Algorithms—ESA '96, Fourth Annual European Symposium, J. Díaz and M. Serna, eds., vol. 1136 of Lecture Notes in Computer Science, Barcelona, Spain, 25–27 Sept. 1996, Springer, pp. 107–120.

[61] P. FERRAGINA, S. MUTHUKRISHNAN, AND M. DE BERG, *Multi-method dispatching: A geometric approach with applications to string matching problems*, in Proc. of the 31st Ann. ACM Symp. on Theory of Computing, Atlanta, Georgia, United States, 1999, ACM Press, pp. 483–491.

[62] R. E. FILMAN, *Polychotomic encoding: A better quasi-optimal bit-vector encoding of tree hierarchies*, in Proceedings of the 16*th* European Conference on Object-Oriented Programming, no. 2374 in Lecture Notes in Computer Science, Malaga, Spain, June 10–14 2002, ECOOP 2002, Springer Verlag, pp. 545–561.

[63] M. FIORE, R. DI COSMO, AND V. BALAT, *Remarks on isomorphisms in typed lambda calculi with empty and sum types*, in Proc. of the 17*th* Annual IEEE Symposium on Logic in Computer Science (LICS'02), July 2002.

[64] M. L. FREDMAN, J. KOMLÓS, AND E. SZEMERÉDI, *Storing a sparse table with $O(1)$ worst case access time*, J. ACM, 31 (1984), pp. 538–544.

[65] H. N. GABOW, J. L. BENTLEY, AND R. E. TARJAN, *Scaling and related techniques for geometry problems*, in Proc. of the 16*th* Ann. ACM Symp. on Theory of Computing, Washington, DC, United States, 1984, ACM Press, pp. 135–143.

[66] J. GIL AND A. ITAI, *The complexity of type analysis of Object Oriented programs*, in Proceedings of the 12*th* European Conference on Object-Oriented Programming, no. 1445 in Lecture Notes in Computer Science, Brussels, Belgium, July 20–24 1998, ECOOP'98, Springer Verlag, pp. 601–634.

[67] J. Y. GIL, *Subtyping arithmetical types*, in 27*th* Symposium on Principles of Programming Languages, POPL'01, London, England, Jan.17–19 2001, ACM SIGPLAN — SIGACT, ACM Press, pp. 276–289.

[68] J. Y. GIL AND P. SWEENEY, *Space- and time-efficient memory layout for multiple inheritance*, in OOPSLA'99 [108], pp. 256–275.

[69] J. Y. GIL AND Y. ZIBIN, *Efficient algorithms for isomorphisms of simple types*. Accepted for publication in Mathematical Structures in Computer Science (MSCS), special issue on Type Isomorphisms.

[70] ——, *Efficient subtyping tests with PQ-encoding*. Accepted for publication in ACM Transactions On Programming Languages And Systems (TOPLAS).

[71] A. GOLDBERG, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Massachusetts, 1984.

[72] R. GRAFL, *CACAO: Ein 64bit JavaVM just-in-time compiler*, master's thesis, University of Vienna, 1996.

[73] R. GUREVIČ, *Equational theory of positive numbers with exponentiation*, American Mathmatical Society, 94 (1985), pp. 135–141.

[74] M. HABIB, Y. CASEAU, L. NOURINE, AND O. RAYNAUD, *Encoding of multiple inheritance hierarchie and partial orders*, Computational Intelligence, 15 (1999), pp. 50–62.

[75] M. HABIB AND L. NOURINE, *Bit-vector encoding for partially ordered sets*, in Bouchitte and Morvan [18], pp. 1–12.

[76] Y. Hollander, M. Morley, and A. Noy, *The e language: A fresh separation of concerns*, in Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems, Zurich, Switzerland, Mar. 12–14 2001, TOOLS 2001 Europe Conference, Prentice-Hall, pp. 41–51.

[77] W. Holst, D. Szafron, Y. Leontiev, and C. Pang, *Multi-method dispatch using single-receiver projections*, Tech. Report TR-98-03, University of Alberta, Edmonton, Alberta, Canada, 1998.

[78] U. Hölzle, C. Chambers, and D. Ungar, *Optimizing dynamically-typed object-oriented languages with polymorphic inline caches*, in Proceedings of the 5^{th} European Conference on Object-Oriented Programming [55].

[79] W. A. Howard, *The formulaes-as-types notion of construction*, in To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, J. R. Hindley and J. P. Seldin, eds., Academic Press, 1980, pp. 479–490.

[80] S. Jha, J. Palsberg, and T. Zhao, *Efficient type matching*, in Proc. of the 5^{th} Foundations of Software Science and Computation Structures, Grenoble, France, April 2002.

[81] S. Jha, J. Palsberg, T. Zhao, and F. Henglein, *Efficient type matching*, TOPPS technical report, DIKU, University of Copenhagen, Universitetsparken 1, 2002. Submitted to Special issue of Higher-Order Symbolic Computation in memoriam Robert Paige.

[82] M. Junger, S. Leipert, and P. Mutzel, *On computing a maximal planar subgraph using PQ-trees*, tech. report, Informatik, Universität zu Köln, 1996.

[83] ——, *A note on computing a maximal planar subgraph using PQ-trees*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 17 (1998), pp. 609–612.

[84] H. Kaci, R. Boyer, P. Lincoln, and R. Nasr, *Efficient implementation of lattice operation*, ACM Trans. Prog. Lang. Syst., 11 (1989), pp. 115–146.

[85] J. Katzenelson, S. S. Pinter, and E. Schenfeld, *Type matching, type-graphs, and the schanuel conjecture*, ACM Trans. Prog. Lang. Syst., 14 (1992), pp. 574–588.

[86] G. Kiczales and L. Rodriguez, *Efficient method dispatch in PCL*, in 1990 ACM Conference on Lisp and Functional Programming, Nice, France, June 1990, ACM, ACM Press, pp. 99–105.

[87] E. Kidd, *Efficient Compression of Generic Function Dispatch Tables*, Tech. Report TR2001-404, Dartmouth College, Computer Science, Hanover, NH, June 2001.

[88] A. Krall, *personal communication*, Feb. 2001.

[89] A. KRALL AND R. GRAFL, *CACAO – a 64 bit JavaVM just-in-time compiler*, in PPoPP'97 Workshop on Java for Science and Engineering Computation, G. C. Fox and W. Li, eds., Las Vegas, June 1997, ACM Press.

[90] A. KRALL, J. VITEK, AND R. N. HORSPOOL, *Near optimal hierarchical encoding of types*, in Proceedings of the 11*th* European Conference on Object-Oriented Programming, no. 1241 in Lecture Notes in Computer Science, Jyväskylä, Finland, June 9-13 1997, ECOOP'97, Springer Verlag, pp. 128–145.

[91] S. LEIPERT, *PQ-trees, an implementation as template class in C++*, tech. report, Informatik, Universität zu Köln, 1997.

[92] A. LEMPEL, S. EVEN, AND I. CEDERBAUM, *An algorithm for planarity testing of graphs*, in Theory of Graphs, International Symposium, New York, NY, 1967, Gordon and Breach, pp. 215–232.

[93] S. B. LIPPMAN, *Inside The C++ Object Model*, Addison-Wesley, 2*nd* ed., 1996.

[94] B. MAGNUSSUN, B. MEYER, AND ET AL., *Who needs need multiple inheritance*. Panel discussion at the European conference on Technology of Object Oriented Programming (TOOLS Europe'94), Mar. 1994.

[95] K. MELHORN AND S. NÄHER, *Dynamic fractional cascading*, Algorithmica, 5 (1990), pp. 215–241.

[96] B. MEYER, *Object-Oriented Software Construction*, International Series in Computer Science, Prentice-Hall, 1988.

[97] ——, *EIFFEL the Language*, Object-Oriented Series, Prentice-Hall, Hemel Hempstead, Hertfordshire, UK, 1992.

[98] D. A. MOON, *Object-oriented programming with flavors*, in OOPSLA'86 [104], pp. 1–8.

[99] W. MUGRIDGE, J. HAMER, AND J. HOSKING, *Multi-methods in a statically-typed programming languages*, in Proceedings of the 5*th* European Conference on Object-Oriented Programming [55].

[100] S. MUTHUKRISHNAN AND M. MÜLLER, *Time and space efficient method-lookup for object-oriented programs*, in Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, New York/Philadelphia, Jan. 28–30 1996, ACM/SIAM, pp. 42–51.

[101] A. C. MYERS, *Bidirectional object layout for separate compilation*, in OOPSLA'95 [106], pp. 124–139.

[102] M. NAIK AND R. KUMAR, *Efficient message dispatch in object-oriented systems*, ACM SIGPLAN Notices, 35 (2000), pp. 49–58.

[103] OOPSLA'01, *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Tampa Bay, Florida, Oct. 14–18 2001, ACM SIGPLAN Notices 36(10) Oct. 2001.

[104] OOPSLA'86, *Proceedings of the 1st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA, Sept. 29 - Oct. 2 1986, ACM SIGPLAN Notices 21(11) Nov. 1986.

[105] OOPSLA'93, *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, USA, Sept. 26 - Oct. 1 1993, ACM SIGPLAN Notices 28(10) Oct. 1993.

[106] OOPSLA'95, *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Austin, Texas, USA, Oct. 15-19 1995, ACM SIGPLAN Notices 30(10) Oct. 1995.

[107] OOPSLA'97, *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Atlanta, Georgia, Oct. 5-9 1997, ACM SIGPLAN Notices 32(10) Oct. 1997.

[108] OOPSLA'99, *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, Nov.1–5 1999, ACM SIGPLAN Notices 34(10) Nov. 1999.

[109] R. PAIGE, *Efficient translation of external input in a dynamically typed language*, in Technology and Foundations—Information Processing 94, B. Pehrson and I. Simon, eds., vol. 1, North-Holland, 1994, pp. 603–608.

[110] K. PALACZ AND J. VITEK, *Java subtype tests in real-time*, in Proceedings of the 17th European Conference on Object-Oriented Programming [54].

[111] J. PALSBERG AND T. ZHAO, *Efficient and flexible matching of recursive types*. Manuscript, 2000.

[112] C. PANG, W. HOLST, Y. LEONTIEV, AND D. SZAFORON, *Multi-method dispatch using multiple row displacement*, in Proceedings of the 13th European Conference on Object-Oriented Programming, no. 1628 in Lecture Notes in Computer Science, Lisbon, Portugal, June 14–18 1999, ECOOP'99, Springer Verlag, pp. 304–328.

[113] W. PUGH AND G. WEDDELL, *Two-directional record layout for multiple inheritance*, in Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, June 1990, ACM SIGPLAN, ACM Press, pp. 85–91. SIGPLAN Notices 25(6).

[114] ——, *On object layout for multiple inheritance*, Technical Report CS-93-22, University of Waterloo—Department of Computer Science, May 1993.

[115] O. RAYNAUD AND E. THIERRY, *A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests*, in Proceedings of the 15th European Conference on Object-Oriented Programming, no. 1850 in Lecture Notes in Computer Science, Budapest, Hungary, June 12–16 2001, ECOOP 2001, Springer Verlag, pp. 165–181.

[116] M. RITTRI, *Retrieving library identifiers via equational matching of types*, in 10th International Conference on Automated Deduction, no. 449 in Lecture Notes in Computer Science, Springer Verlag, July 1990, pp. 603–617.

[117] ——, *Using types as search keys in function libraries*, Journal of Functional Programming, 1 (1991), pp. 71–89.

[118] A. ROYER, *Optimizing Method Search with Lookup Caches and Incremental Coloring*, in Proceedings of the 7th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Vancouver, British Columbia, Canada, Oct.18-22 1992, OOPSLA'92, ACM SIGPLAN Notices 27(10) Oct. 1992, pp. 110–126.

[119] M. A. SCHUBERT, P. L.K., AND J. TAUGHER, *Determining type, part, colour, and time relationships*, Computer, 16 (special issue on Knowledge Representation) (1983), pp. 53–60.

[120] A. SHALIT, *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*, Addison-Wesley, Reading, Mass., 1997.

[121] D. SLEATOR AND R. TARJAN, *Self-adjusting binary search trees*, J. ACM, 32 (1985), pp. 652–686.

[122] S. V. SOLOVIEV, *The category of finite sets and cartesian closed categories*, Journal of Soviet Mathematics, 22(3) (1983), pp. 1387–1400.

[123] B. STROUSTRUP, *The Design and Evolution of C++*, Addison-Wesley, Reading, Massachusetts, Mar. 1994.

[124] ——, *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 3rd ed., 1997.

[125] A. TARSKI, *A Decision Method for Elementary Algebra and Geometry*, University of California Press, Berkeley, CA, 2nd ed., 1951.

[126] W. T. TROTTER, *Combinatorics and Partially Ordered Sets: Dimension Theory*, The Johns Hopkins University Press, 1992.

[127] M. F. VAN BOMMEL AND T. J. BECK, *Incremental encoding of multiple inheritance hierarchies*, in Proceedings of the 8th International Conference on Information Knowledgement (CIKM-99), N.Y., Nov. 2–6 2000, ACM Press, pp. 507–513.

[128] P. VAN EMDE BOAS, *Preserving order in a forest in less than logarithmic time and linear space*, Inf. Process. Lett., 6(3) (1977), pp. 80–82.

[129] P. VAN EMDE BOAS, R. KAAS, AND E. ZIJLSTRA, *Design and implementation of an efficient priority queue*, Math. Systems Theory, 10 (1977), pp. 99–127.

[130] J. VITEK, *Compact dispatch tables for dynamically typed programming languages*, master's thesis, University of Victoria, 1995.

[131] J. VITEK AND R. N. HORSPOOL, *Taming message passing: Efficient method lookup for dynamically typed object-oriented languages*, in Proceedings of the 8th European Conference on Object-Oriented Programming [56].

[132] ——, *Compact dispatch tables for dynamically typed object oriented languages*, in Compiler Construction, 6th International Conference, T. Gyimothy, ed., vol. 1060 of Lecture Notes in Computer Science, Linköping, Sweden, 24–26 Apr. 1996, Springer, pp. 309–325.

[133] J. VITEK, R. N. HORSPOOL, AND A. KRALL, *Efficient type inclusion tests*, in OOPSLA'97 [107], pp. 142–157.

[134] D. E. WILLARD, *New trie data structures which support very fast search operations.*, J. Comput. Sys. Sci., 28 (1984), pp. 379–394.

[135] O. ZENDRA, C. COLNET, AND S. COLLIN, *Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler*, in OOPSLA'97 [107], pp. 125–141.

[136] Y. ZIBIN AND J. Y. GIL, *Efficient subtyping tests with PQ-encoding*, in OOPSLA'01 [103], pp. 96–107.

[137] ——, *Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching*, in Proceedings of the 17th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Seattle, Washington, Nov. 4–8 2002, OOPSLA'02, ACM SIGPLAN Notices 37(10) Nov. 2002, pp. 142–160.

[138] ——, *Incremental algorithms for dispatching in dynamically typed languages*, in Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'03), ACM Press, 2003, pp. 126–138.

[139] ——, *Two-dimensional bi-directional object layout*, in Proceedings of the 17th European Conference on Object-Oriented Programming [54], pp. 329–350.

[140] Y. ZIBIN, J. Y. GIL, AND J. CONSIDINE, *Efficient algorithms for isomorphisms of simple types*, in Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'03), ACM Press, 2003, pp. 160–171.