Final Thesis

Analysis of a multiple dispatch algorithm

by

Johannes Holmberg

LITH-IDA-EX--04/018--SE

2004-02-17

Final Thesis

Analysis of a multiple dispatch algorithm

by

Johannes Holmberg

LiTH-IDA-EX--04/018--SE

2004-02-17

Linköpings universitet

Supervisor: Björn Hägglund,
Department of Computer and Information Science

Examiner: Anders Haraldsson,
Department of Computer and Information Science

Abstract

The development of the new programming language Scream, within the project Software Renaissance, led to the need of a good multiple dispatch algorithm. A multiple dispatch algorithm, called Compressed n-dimensional table with row sharing; CNT-RS, was developed from the algorithm Compressed n-dimensional table, CNT. The purpose of CNT-RS was to create a more efficient algorithm. This report is the result of the work to analyse the CNT-RS algorithm.

In this report the domain of multiple dispatch, the multiple dispatch algorithm CNT and the new extended algorithm CNT-RS are presented. The correctness of CNT-RS algorithm is shown and it's proven that the CNT-RS algorithm is at least as good as the CNT algorithm, in regards to space complexity of the dispatch structure.

Table of contents

1 Introduction

In this chapter an introduction to this thesis is given.

1.1 Background

This master thesis was made within the Software Renaissance project [8]. “The Software Renaissance Project aims at an integrated and extremely adaptable programming environment for interactive intentional programming. Ideally, adaptability is taken so far that language features are provided in libraries rather than languages, in order to facilitate third party development and deployment of individual language concepts. Indeed, the whole purpose with adaptability is to maximize the speed with which the programming environment evolves towards new heights for each problem domain.” [8]

1.1.1 Scream

In the Software Renaissance project [8] a new multiple dispatch language, Scream [6], is being created. A large portion of the execution of programs written in Scream will be in the dispatch so a good, in regard to time and space complexity, multiple dispatch algorithm will be needed.

The creation of Scream aims at creating a language with such an adaptability and expressive power that the problem of language disintegration gets solved.

1.2 Dispatch

Dispatching is the determination of which method to use when a function call is made. This can be done statically at compilation time, if possible, or dynamically at run time. To be able to do it statically

the type hierarchy and the methods has to be static and we must have a strongly typed language so that we know the type of any object at all places. There are also different kinds of mixes of dispatching statically and dynamically.

Today's objectoriented languages use *single dispatch* to determine which method to be invoked at a function call. In the single dispatch scheme the dispatch is based on the method name and one single argument, one single receiver or the object the function is working on.

In object-orientation single dispatch is used in the way that all methods are associated with one type. For example when a function call $o_1.m(o_2,...,o_n)$ is made, the method used is determined by the type of object o_1. The determination is made by checking the type of o_1, let's say that it's T_1, and if T_1 has a method m that method is used. If T_1 doesn't have a method m one goes on and checks if any of the parents of T_1 in the type hierarchy has any method m. One goes on like this until a method m is found. There are more efficient ways to do this than the one above. One could for example create a table with the associations of method name and type to which method to use. At a function call only a simple table look up is needed. The method used will be the method that belongs to the type that is the most *specific* applicable. That is to say the applicable type that is the subtype of all other applicable types.

This scheme has its limitations. For example when a method is used in a relation between two or more objects one would like that the method used is dependent on all the types in the relation. Otherwise one has to associate the method with one type and in the implementation for the method have a case separation for the rest of the arguments. This could obscure the view of what the method actually does.

As one can see from the way the method is found, by looking at the parent type of the type if there is no method for the type, if only single inheritance is allowed there will always be one most specific, if any, method. If we allow multiple inheritance we could get an ambiguity of which method that are the most specific if two sibling types both have the method. So if one have single inheritance and multiple inheritance one needs a scheme for solving ambiguous function calls.

1.3 Multiple dispatch

With multiple dispatch the dispatch is made on some or all arguments. For example if a function call $m(o_1,...,o_n)$ is made, the method to use could be decided on the basis of the types of all objects $o_1,...,o_n$. Let's say that they are $T_1,...,T_n$. The dispatch will look for a method m the formal arguments $T_1,...,T_n$. If such a method is found, it will be used, otherwise a method with formal arguments $T_1',...,T_n'$ such that $T_i \le T_i'$ will be looked for. The method used will be the most specific one.

Thanks to this dependent on many arguments multiple dispatch is more powerful than single dispatch, but also more complex.

1.3.1 The Visitor design pattern

A consequence of the more powerful expressive power, from multiple dispatch, is that the design pattern Visitor [4: p331-344] becomes superfluous. A design pattern is a core of the solution to a recurring design an implementation problem. When you use a design pattern you take this core and adapt it to the current problem. These design patterns have been used for a long time and have evolved over many years. The main motivation for documenting these design patterns was to get designing of reusable object-oriented software. Design patterns are however not limited to object-oriented software design and

implementation in object-oriented languages. Procedural languages would need design patterns for “Inheritance”, “Encapsulation” and “Polymorphism, things that are built-in in object-oriented languages. In the same way object-oriented languages would need the Visitor pattern, but the key to the visitor pattern, double dispatch, is a part of multiple dispatch languages. The double dispatch the Visitor pattern is using is that the operation that gets executed depends on both the type of Visitor and the type of Element it visits.

The motivation for the Visitor pattern is that you want to add operations to classes without changing them. This is done through that the Visitor yield objects whose only responsibilities are to implement a request on another object or group of objects. The design patterns can be classified by their purpose and scope. The Visitor pattern is classified as Behavioural purpose and Object scope.

As the Visitor pattern has been documented it has to have been occurring often enough to be worth it and therefore there exist a need for the multiple dispatch. You can say that the Visitor pattern exists because of the lack of multiple dispatch.

1.3.2 Multiple dispatch today

A reason for that the multiple dispatch isn’t used today is that people have thought that the added time or space complexity isn’t compensated by the greater expressive power of the multiple dispatch. This excuse gets less and less relevant as the algorithms for multiple dispatch gets better and better in time or space complexity. One argument for using the more complex multiple dispatch is that designers and programmers should do what they do best, i.e. design and implement, and not be trying to come up with workarounds because of limitations in the language. Also these workarounds can make the program execute slowly. In these cases a good implementation of multiple dispatch into the language would generate

the possibility to write more easily understandable code and a faster executing program.

There are some techniques to perform the multiple dispatch. Two of the proposed techniques are Single receiver projections, SRP [5], and Multiple row displacement, MRD [7]. SRP algorithm means that you have n single-receiver dispatch tables and projects n-arity methods on these n tables. For good computation time and space requirements the algorithm is implemented with bit vectors and needs the aid of special hardware support for some bit operations. The MRD do the same elimination and grouping as the CNT and by row displacement combines it into one vector and thereby can make some more savings in space requirement.

There are three main concerns, regarding the efficiency of a multiple dispatch algorithm. Those concerns are the time to create any dispatch structure, the size of that structure and the time to do the method dispatching at a function call.

Dispatch can be performed in constant time using dispatch tables. The tables, if not compressed, become very large. Consider that a programming library can contain a couple of hundred types and an application using this library can have something like a hundred types. Let's say for example that we have, in total, a type hierarchy Θ with the number of types $|\Theta| = 800$ then the approximate size of an uncompressed dispatch table for a three-targeted method would be $|\Theta|^3 \approx 500\text{MB}$. The corresponding sizes for a four-targeted and five-targeted method are respectively 400GB and 300TB. Note that this is for one generic function. As shown by these numbers to use uncompressed dispatch tables are not possible.

1.4 Assignment

The assignment was to study different aspects of the new algorithm CNT-RS. At the beginning the focus was at computing the amortised space complexity of the multiple dispatch algorithm CNT-RS and comparing it to the multiple dispatch algorithm CNT. The computation and definition of this amortised space complexity dragged out on time so the focus of the work shifted to prove the correctness of the CNT-RS algorithm.

A major part of the work was to describe the two algorithms and especially the newly developed CNT-RS algorithm as it hasn't been described in writing before.

1.5 Organisation of this report

The organisation of this report is as follows: We start of with an introduction containing the background, to this report, dispatch and multiple dispatch, and the assignment in chapter 1 Introduction. It is followed by the description of the two algorithms, CNT and CNT-RS, in chapter 2 The algorithms. Both algorithms' descriptions are accompanied with an example being walked trough. Also the CNT-RS algorithm is proven to be correct. In chapter 3 Space complexity, the aspect of space complexity is handled. The space complexity of the algorithms is described and compared. It's also shown that the CNT-RS also can give a worst case with uncompressed dispatch structure. Following this, possible future work is described in chapter 4 Future work. And finishing this report is chapter 5 Conclusion, where the conclusions of this report are drawn.

2 The algorithms of CNT and CNT-RS

In this chapter the CNT and the CNT-RS algorithms are presented and applied to an example. Also the correctness of the CNT-RS algorithm is shown.

2.1 Some notions used

If nothing else is explicitly stated the following denotations is used.

- Θ denotes the type hierarchy.
- $|\Theta|$ denotes the number of types in the type hierarchy.
- m denotes the generic function.
- n denotes the arity of the of the generic function.
- T denotes a type.
- T_i denotes a type at argument position i.
- $MS(\ )$ denotes the most specific method, the method to dispatch to.
- $Pole_m^i$ denotes the i-poles. $|Pole_m^i|$ denotes the number of i-poles.
- $MPole_m^i$ denotes the i-multipoles. $|MPole_m^i|$ denotes the number of i-multipoles.

2.2 CNT - Compressed n-dimensional table

Compressed n-dimensional table [3]. As the name implies the CNT algorithm uses n-dimensional tables for dispatching and one dispatch table per generic function with arity n. The CNT algorithm builds on two observations: (i) that entries corresponding to invocations for which there are no applicable method can be eliminated and (ii) that identical (n-1)-dimensional rows can be grouped together. The naive approach, to first build the uncompressed dispatch table and then compress it, has a worst-case time complexity of $O(n*|\Theta|^{n+1})$ per generic function, which is not acceptable performance. Also the size

of the uncompressed table will at an early state be too large to handle. The idea of the CNT algorithm is to determine which entries can be eliminated and which can be grouped without having to go through the whole dispatch table and thereby avoid the bad time complexity of the naive approach. There are two simple conditions to determine when elimination and grouping can be done. They are as follow:

Elimination Condition: The entry for type T in the ith dimension of the dispatch table of m can be eliminated if and only if $\forall(T_1,...,T_{n-1}) \in \Theta^{n-1}, MS(m(T_1,...,T_{i-1},T,T_i,...,T_{n-1})) = \emptyset$.

Grouping Condition: The two entries for types T and T' in the ith dimension of the dispatch table of m can be grouped if and only if $\forall(T_1,...,T_{i-1},T_{i+1},...,T_n) \in \Theta^{n-1}$,

$MS(m(T_1,...,T_{i-1},T,T_{i+1},...,T_n)) = MS(m(T_1,...,T_{n-1},T',T_{n+1},...,T_n))$.

This in words is: If there is no most specific method for method m when type T is at the ith position, the entries can be eliminated. If most specific method for method m is the same when the ith position is of type T and T' they can be grouped.

All the groups created by groping contain a supertype of all the other types in the group. This type is called a *pole* and the types in the group are said to belong to the *influence* of the pole. A type that does not belong to an influence can be eliminated. A pole is called an *i-pole* if it's a pole for the ith argument position. The set of i-poles for the generic function m is denoted by $Pole_m^i$. If a type belongs to the formal arguments of the method m the pole is a primary pole. All other poles are secondary poles. The poles form a pole hierarchy in accordance with the type hierarchy.

These poles can be computed in a single pass over the set of types using the notion of $closest - poles_m^i$.

Definition 2.2.1. Let m be a generic function of arity n, $T \in \Theta$, and $1 \le i \le n$; the set of closest poles of T is

$$closest - poles_m^i(T) = \min_{\le}\{T' \in Pole_m^i \mid T < T'\}$$

How you use this is to make a total order of the partially ordered type hierarchy such that all parents come before their children. In this total order you go from the start and associate each type with the pole in which influence the type is in. Which pole this is, is decided by these conditions:

If the type is a primary pole the type is associated with itself.
If it's not a primary pole:
If the type has no parents or the parents are not associated with any pole the type is not in any pole's influence and the type isn't associated with anything.
Else if any of the type's parents' associated poles is the subtype of all the other associated poles, the type is associated with this pole.
Else the type is associated with itself; the type is a secondary pole.

With this approach to computing the poles you don't get the optimal compression of the dispatch table, but it's an easy way to compute the poles and you don't do any compressions that are not allowed. The compression you are missing is that you can get more than the optimal number of secondary poles because types, which are not primary poles, inheriting from exactly the same primary poles are both marked as secondary poles even though they could be grouped together according to the grouping condition.

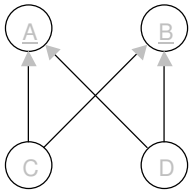


Figure 1 Illustrating missing compression.

Figure 1 illustrates this missing of possible compression, where A and B are poles and C and D will be different poles even though they could be grouped together.

2.2.1 Example: An example with CNT.

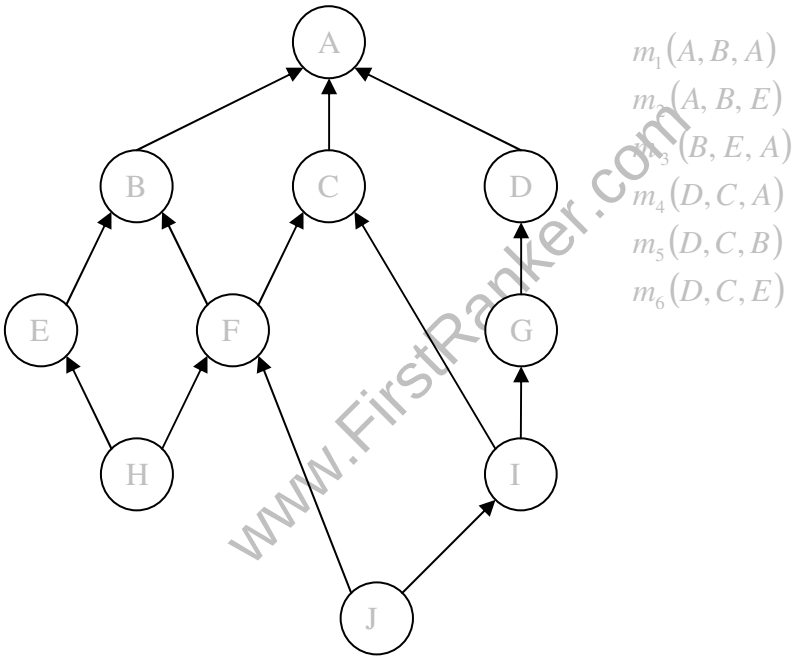


Figure 2 The example problem

A to J are types in the type hierarchy, a type is a subtype of an other type if there is a arrow from the first type to the other and *m* is a generic function with arity 3. We start of with finding the primary poles for each argument position. They are at the first position {A,B,D}, at the second {B,C,E} and at the third{A,B,E}. Next we compute the association of each type to the pole in which's influence the type belong. First we make a total order of the type hierarchy so that we can go through it in a way so that we visit the supertypes before the subtypes. We use A to J alphabetically ordered. We begin with the first argument position.

A	A
B	B
C	A
D	D
E	B
F	B
G	D
H	B
I	D
J	J

Figure 3 Table of type to pole associations

Then we begin with A and A is a primary pole so we associate it with itself. Next B and B is also a primary pole so it's also associated with itself. Next C and C is not a primary pole. C has A as only parent so C is associated with the same as A, A. Next D and D is a primary pole so it's associated with it self. Next E and E has B as only parent so E is associated with the same as B, B. Next F and F has B and C as

parents. B and C are associated with B and A and as B is a subtype of A F is associated with B. Next G and G has as its only parent D so G is associated with the same as D, D. Next H and H has E and F as its parents. E and F are both associated with B so H is also associated with B. Next I and I has C and G as its parents. C and G are associated with A and D and as D is a subtype of A I is associated with D. Last J and J has as its parents F and I. F and I are associated with B and D and as neither is the subtype of the other J is associated with itself.

We can now see that the poles in the first argument position is {A,B,D,J}. Also it can be seen that {A,C} is in the influence of A, {B,E,F,H} is in the influence of B, {D,G,I} is in the influence of D and {J} is in the influence of J.

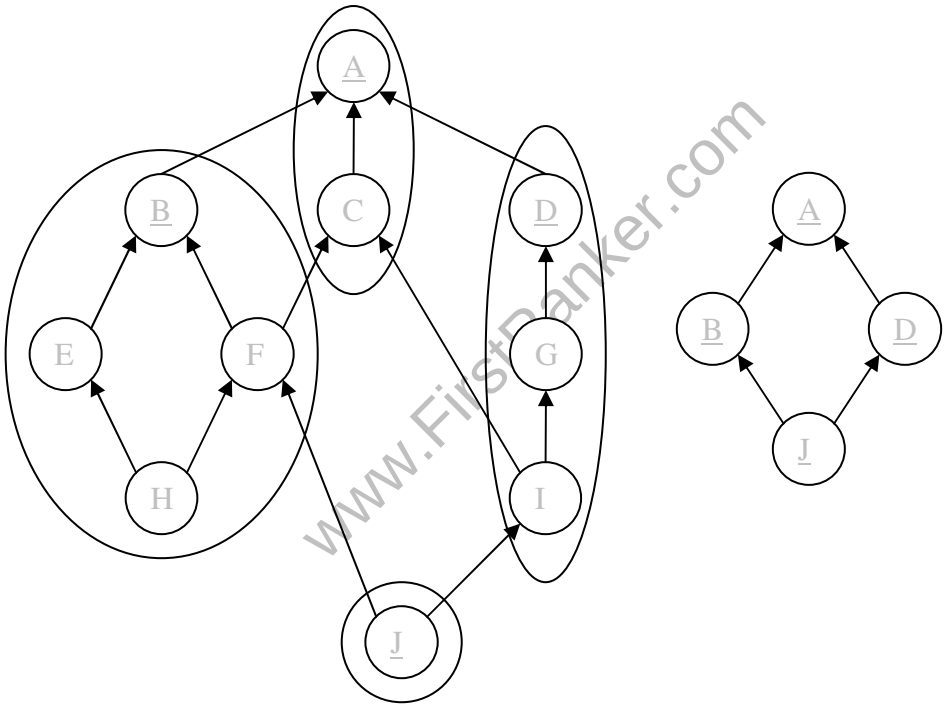


Figure 4 The type hierarchy with poles and influences marked and the pole hierarchy

The poles marked with an underline and the influences with an oval. On the right we have the pole hierarchy for the first argument position. In the same way the pole hierarchies for the second and third argument position is computed and we get the following.

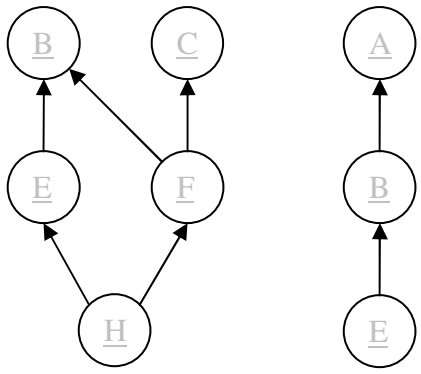


Figure 5 The second and third pole hierarchies

Now when we have a pole hierarchy for each argument position we start building the dispatch table. For the first argument position we need a vector of size four for the poles, for the second four vectors of size five and for the third twenty vectors of size three.

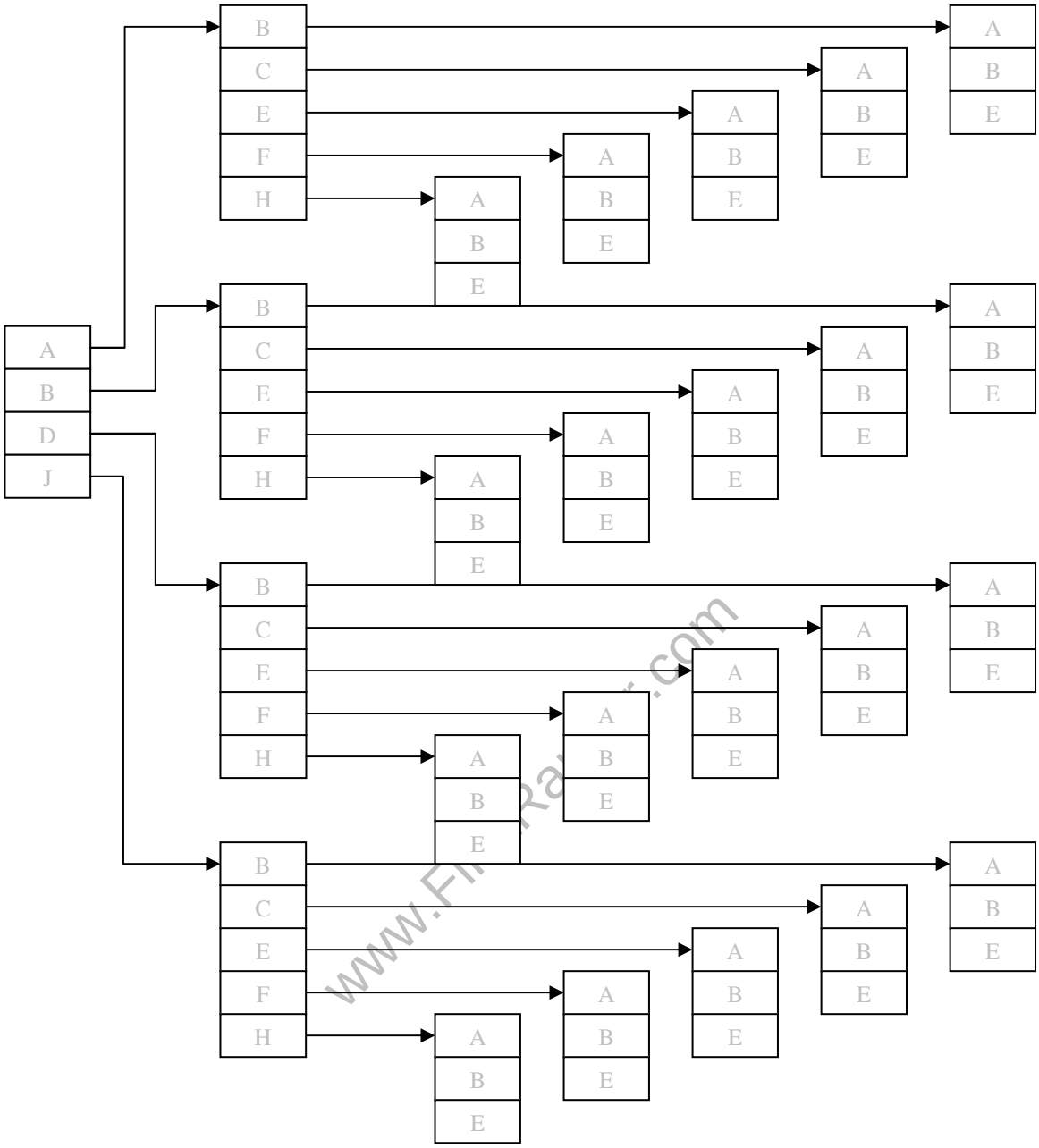
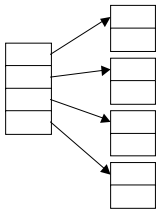


Figure 6 The dispatch table for the example problem

The number of cells used is $4+4*5+4*5*3=84$, too be compared with $10+10*10+10*10*10=1110$ for the uncompressed table.

2.3 CNT-RS - Compressed n-dimensional table with row sharing



Compressed n-dimensional table with row sharing. Even though the name of the algorithm contains n-dimensional table I personally thinks it's easier to visualise the dispatch structure as a collection of vectors, so elements of the dispatch table will below be referred to as vectors.

The CNT-RS algorithm is an extension of the CNT algorithm. This extension is to look at more than one argument at a time and the introduction of the notion of *multipole*. The CNT-RS algorithm starts the same way as the CNT by computing all the poles for each generic function m and for each argument position i of m. Then it goes on to computing the multipoles. From the first and second pole hierarchy a new multitype hierarchy is created. Combining every type from the first pole hierarchy with every type from the second pole hierarchy creates the multitypes. The hierarchy is created by the following sub-/supertype condition: $(A.B)\leq(C.D) \Leftrightarrow A\leq C \wedge B\leq D$. From this new multitype hierarchy using the following slightly modified elmination and grouping conditions create a multipole hierarchy.

The elimination and grouping condition:

Elimination Condition: The vector for $T\in\Theta^k$ in the dispatch table of m can be eliminated if

$$\forall(T_{k+1},...,T_n)\in\Theta^{n-k}, MS(m(T,T_{k+1},...,T_n))=\emptyset. \quad (1)$$

Grouping Condition: The vectors for $T \in \Theta^k$ and $T' \in \Theta^k$ in the dispatch table of m are identical and can be shared if

$$\forall (T_{k+1},...,T_n) \in \Theta^{n-k}, MS(m(T,T_{k+1},...,T_n)) = MS(m(T',T_{k+1},...,T_n)). \quad (2)$$

After this we go on and create a new multitype hierarchy from the newly created multipole hierarchy and the third pole hierarchy and from that compute the multipole hierarchy and so on through all the arguments.

2.3.1 Example: Same example with CNT-RS.

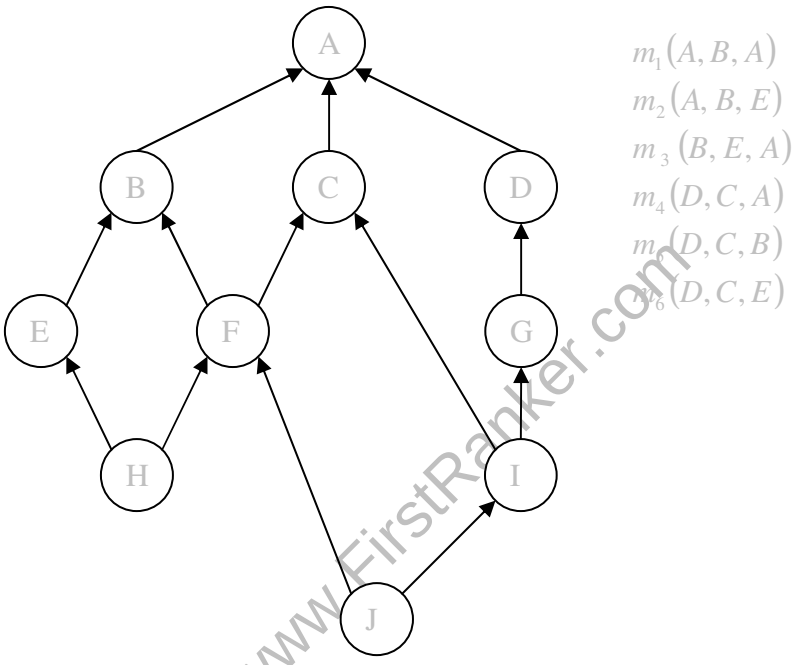


Figure 7 The example problem

Now we go through the same example again for CNT-RS. First we compute a pole hierarchy for each argument position. This we done in the CNT example and here is a repetition of them.

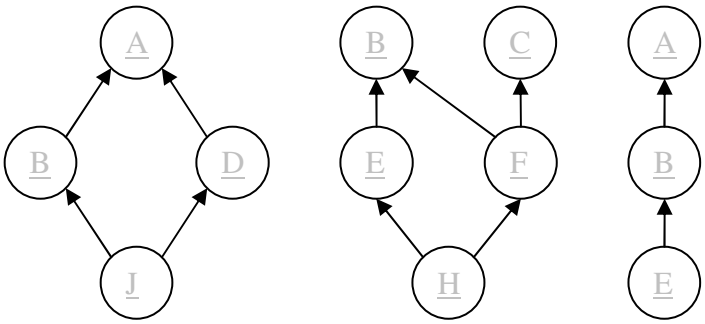


Figure 8 The pole hierarchies for the first, second and third argument position

Now when we have the pole hierarchies for each argument position we begin with the computation of the multipole hierarchies. The first multipole hierarchy is the same as the one for the first argument position. To compute the second multipole hierarchy we start of by creating the new multitype hierarchy from the first multipole hierarchy and the second pole hierarchy.

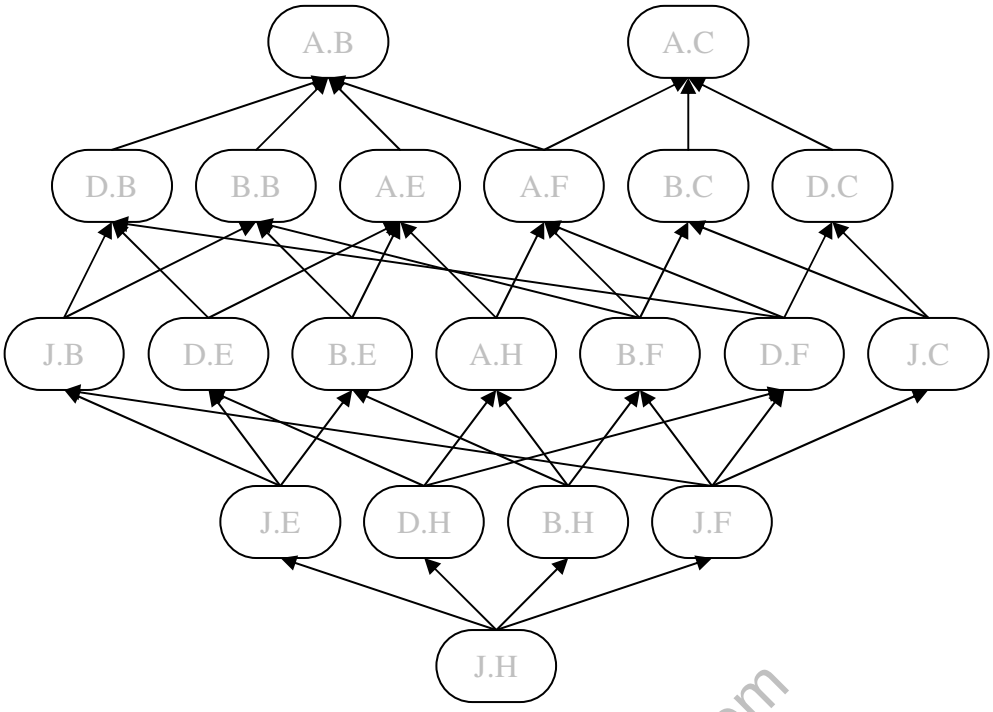


Figure 9 The multitype hierarchy made from the first and second pole hierarchies

From the methods we note that the primary multipoles are {(A.B),(B.E),(D.C)}. From the new multitype hierarchy and the primary multipoles we compute the multipole hierarchy in the same way as computing the poles above.

(A.B)	(A.B)	(B.B)	(A.B)	(D.B)	(A.B)	(J.B)	(A.B)
(A.C)	-	(B.C)	-	(D.C)	(D.C)	(J.C)	(D.C)
(A.E)	(A.B)	(B.E)	(B.E)	(D.E)	(A.B)	(J.E)	(B.E)
(A.F)	(A.B)	(B.F)	(A.B)	(D.F)	(D.F)	(J.F)	(D.F)
(A.H)	(A.B)	(B.H)	(B.E)	(D.H)	(D.F)	(J.H)	(J.H)

Figure 10 Table of multitype to multipole associations

One thing worth noting here is that (A.C) and (B.C) are not associated to any multitype and can be eliminated. We get the following second multipole hierarchy.

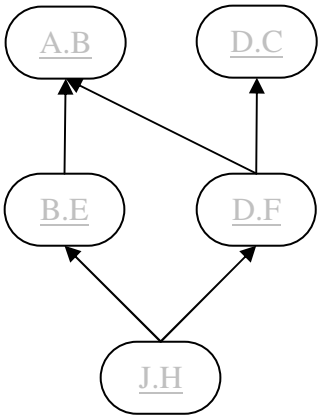


Figure 11 The second multipole hierarchy

In other words the vectors for (A.C) and (B.C) can be eliminated, (A.B), (A.E), (A.F), (A.H), (B.B), (B.F), (D.B), (D.E) and (J.B) can be shared, (B.E), (B.H) and (J.E) can be shared, (D.C) and (J.C) can be shared, (D.F), (D.H) and (J.F) can be shared and lastly (J.H) can't be eliminated or shared with anyone.

The last multipole hierarchy is not used for compressing the dispatch structure but for knowing which method to dispatch to. We also get some additional information that can be used to help solve ambiguities and rejections. So let's compute the last multipole hierarchy. In the same way as before first create a new multitype hierarchy, note the primary multipoles and compute the influences.

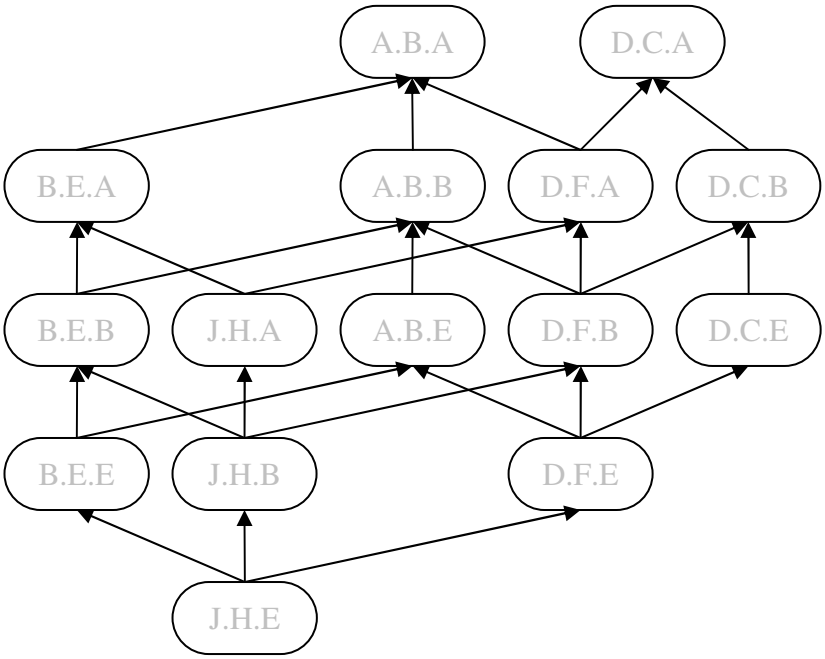


Figure 12 The third multitype hierarchy

The primary multipoles are (A.B.A), (A.B.E), (B.E.A), (D.C.A), (D.C.B) and (D.C.E). As these are the same as the formal signatures of the methods they will represent the methods that will be dispatched to.

(A.B.A)	(A.B.A)
(A.B.B)	(A.B.A)
(A.B.E)	(A.B.E)
(D.C.A)	(D.C.A)
(D.C.B)	(D.C.B)
(D.C.E)	(D.C.E)

(B.E.A)	(B.E.A)
(B.E.B)	(B.E.A)
(B.E.E)	(B.E.E)
(D.F.A)	(D.F.A)
(D.F.B)	(D.F.B)
(D.F.E)	(D.F.E)

(J.H.A)	(J.H.A)
(J.H.B)	(J.H.B)
(J.H.E)	(J.H.E)

Figure 13 Table of multitype to multipole associations

The secondary multipoles will represent a conflict or ambiguity in the dispatch.

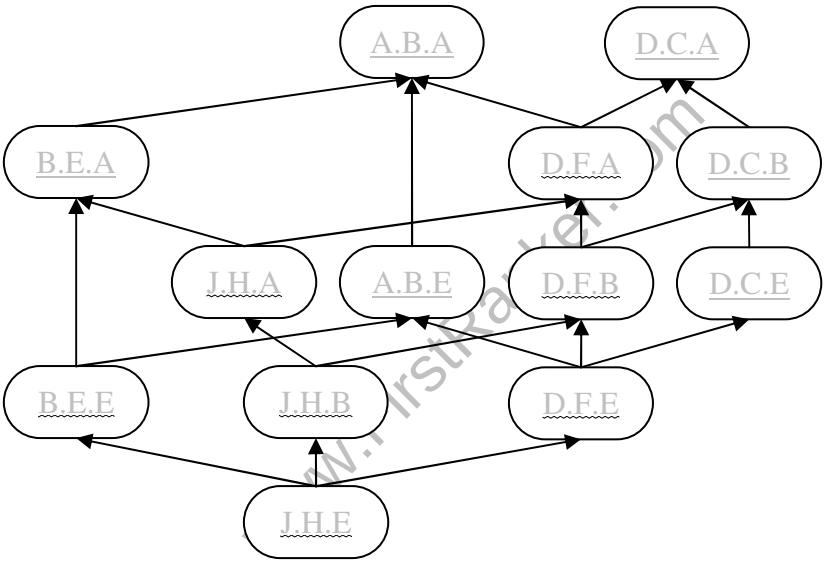


Figure 14 The last multipole hierarchy

Primary multipoles marked with straight underline and secondary multipoles marked with wavy underline. Now let's build the dispatch structure.

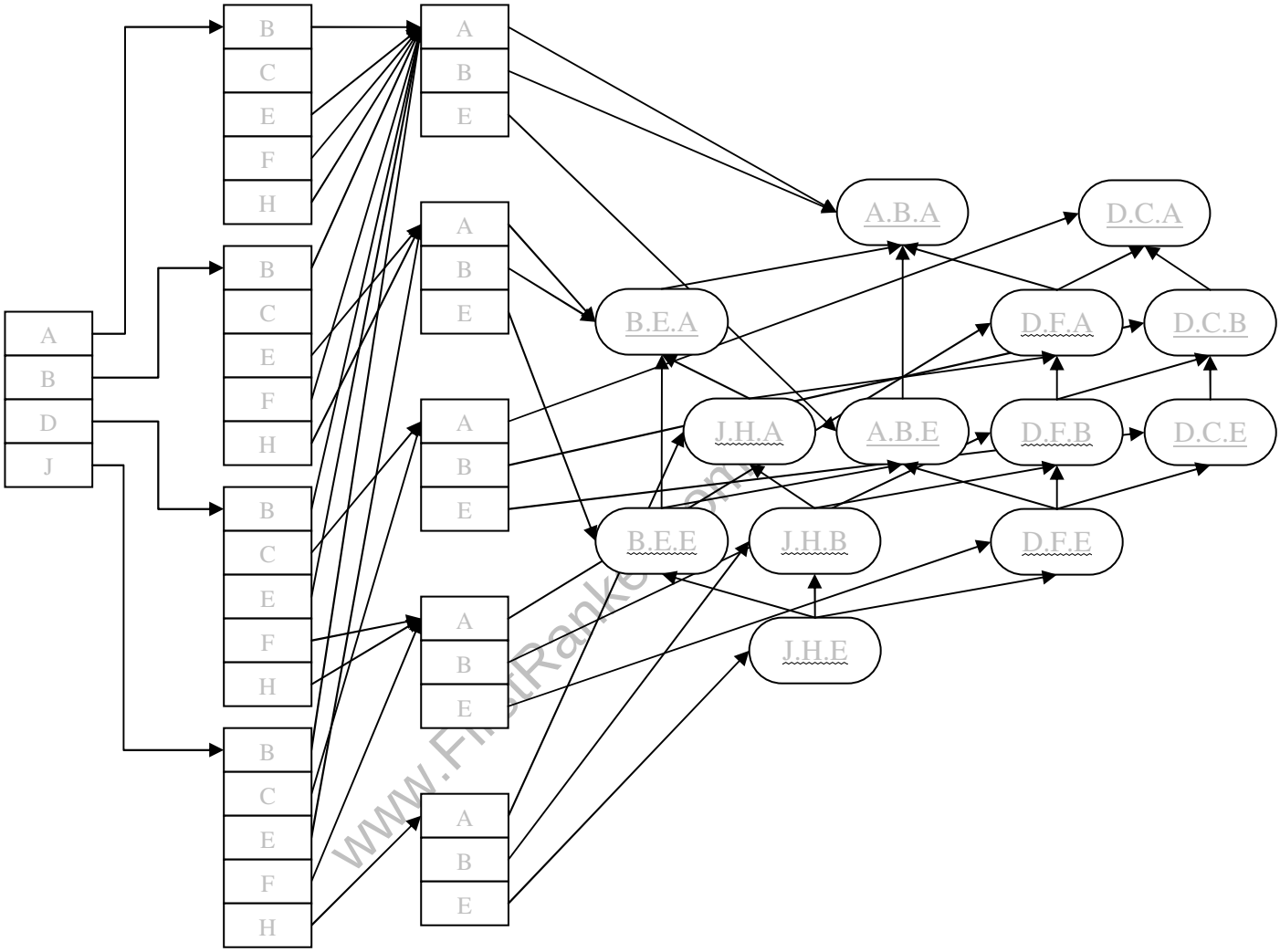


Figure 15 The dispatch structure

The number of cells used is $4+4*5+5*3=39$ too being compared with 84 for CNT and 1110 for the uncompressed table.

2.3.2 Correctness of CNT-RS

The correctness proof will be given in some what analogy with the correctness proof of the CNT algorithm in [3]. This analogy should make it easier for readers, who are already familiar with the CNT proof, to digest this without risking the understanding for readers who don't. First some definitions will be made and some facts will be stated. All definitions are adaptations of the definitions in [3] to the "multi case". At all of these a reference to the analogue of it in [3] will be given.

Definition 2.3.2.1. (Definition 2.1.) A method $m_k(T_k, T_k^{i+1},...,T_k^n)$ is applicable to a signature $(T, T^{i+1},...,T^n)$, denoted by $m_k \geq (T, T^{i+1},...,T^n)$, if and only if $(T_k, T_k^{i+1},...,T_k^n) \geq (T, T^{i+1},...,T^n)$.

Definition 2.3.2.2. (Definition 4.1.3.) The ith static multiarguments of a generic function m, denoted $MStatic_m^i$, are the multitypes of the i first formal arguments of the methods of m:
$MStatic_m^i = \{T \in \Theta^i \mid \exists m(T, T_{i+1},...,T_n)\}$

Definition 2.3.2.3. (Definition 4.1.4.) The ith dynamic multiarguments of a generic function m, denoted $MDynamic_m^i$, are the cover of the ith static multiarguments of m: $MDynamic_m^i = cover(MStatic_m^i)$, where $cover(T) = \{T' \mid T' \leq T\}$
They represent the multitypes that can appear at the first argument positions in invocations of m at run-time. $MDynamic_m$ is the same as $MDynamic_m^n$ and $MDynamic_m \subseteq MDynamic_m^k \times \prod_{i=k+1}^{n} Dynamic_m^i, k \in [0,n]$.

Fact 2.3.2.4. (Fact 4.1.5.) $s \in MDynamic_m \Leftrightarrow applicable(m(s)) \neq \emptyset$.

Fact 2.3.2.5. (Fact 4.1.7.) $MPole_m^i \subset MDynamic_m^i$.

Definition 2.3.2.6. (Definition 4.1.13.) With every i-multipole T of a generic function m is associated the set of subtypes of T, noted $Influence_m^i(T)$, which is defined as

$Influence_m^i(T) = \{T' \leq T \mid \forall T'' \in MPole_m^i, T' \not\leq T'' or T \leq T''\}$.

Fact 2.3.2.7. (Proposition 4.1.15.) Given a generic function m of arity n, and $i \in \{1,...,n\}$, let $MPole_m^i = \{T_1,...,T_l\}$, then

$\bigcup_{k=1}^{l} Influence_m^i(T_k) = MDynamic_m^i$.

Definition 2.3.2.8. (Definition 4.1.16.) For each type T in Θ^i, we define $Mpole_m^i$:

If $T \in MDynamic_m^i$ then $Mpole_m^i(T) = T' \Leftrightarrow T \in Influence_m^i(T')$

Otherwise $Mpole_m^i(T) = 0$

2.3.2.1 Correctness of the elimination condition

The following statement will be proven: For every generic function m of arity n, and $i \in \{1,...,n\}$, the vector for multitype T can be eliminated from the dimension i of the dispatch table of m if T does not belong to the influence of any i-multipole.

We prove that if we assume that T does not belong to the influence of any i-multipole, then it verifies $\forall (T_{i+1},...,T_n) \in \Theta^{n-i}$, $MS(m(T,T_{i+1},...,T_n)) = \emptyset$.

By Fact 2.3.2.7, $T \notin MDynamic_m^i$. By Definition 2.3.2.3, $(T,T_{i+1},...,T_n) \notin MDynamic_m$. By Fact 2.3.2.4, $applicable(m(T,T_{i+1},...,T_n)) = \emptyset$. Hence as there are can be no most specific applicable method when there are no applicable method we get $MS(m(T,T_{i+1}...,T_n)) = \emptyset$.

As the cell associated with $(T,T_{i+1},...,T_n)$ is composed of members of $MS(m(T,T_{i+1},...,T_n))$, it is also empty. Consequently the entry for T can be eliminated.

2.3.2.2 Correctness of the grouping condition

Let m be a generic function of arity n, for each i, $1 \le i \le n$, and for each multitype $T \in MDynamic_m^i$, the entries for multitype T and $Mpole_m^i(T)$ can be grouped in the dispatch table.

We prove that for each i, $1 \le i \le n$, $\forall T \in MDynamic_m^i$, $\forall (T_{i+1},...,T_n) \in \Theta^{n-i}$, we have

$$MS(m(T,T_{i+1},...,T_n)) = MS(m(Mpole_m^i(T),T_{i+1},...,T_n)). \quad (7)$$

If there exist j, $j > i$, such that $T_j \notin Dynamic_m^i$ then $(T,T_{i+1},...,T_n) \notin MDynamic_m$, and by Fact 2.3.2.4, both sides of (7) are $\emptyset$.

Assuming now that $(T,T_{i+1},...,T_n) \in MDynamic_m$, to show (7) we prove that $applicable(m(T,T_{i+1},...,T_n)) = applicable(m(Mpole_m^i(T),T_{i+1},...,T_n))$ and as the most specific is decided by the type hierarchy, that is all needed to be shown.

$\subseteq$: As $(T,T_{i+1},...,T_n) \in MDynamic_m$, $applicable(m(T,T_{i+1},...,T_n)) \neq \emptyset$. Let $m_k(T_k,T_k^{i+1},...,T_k^n) \in applicable(m(T,T_{i+1},...,T_n))$. From Definition 2.3.2.1, $T \le T_k$ and T_k is an i-multipole. As $Mpole_m^i(T) \le T_k$, m_k is applicable to $(Mpole_m^i(T),T_{i+1},...,T_n)$.

$\supseteq$: As $applicable(m(T,T_{i+1},...,T_n)) \neq \emptyset$ and $applicable(m(T,T_{i+1},...,T_n)) \subseteq applicable(m(Mpole_m^i(T),T_{i+1},...,T_n))$, we get $applicable(m(Mpole_m^i(T),T_{i+1},...,T_n)) \neq \emptyset$. Let $m_k \in applicable(m(Mpole_m^i(T),T_{i+1},...,T_n))$ From definition 2.3.2.1, we have $Mpole_m^i(T) \le T_k$, and we also have $T \le Mpole_m^i(T)$. By

transitivity, $T \leq T_k$, and by Definition 2.3.2.1, $m_k \in applicable(m(T, T_{i+1}, ..., T_n))$.

This proves (7), the entries associated with T and $Mpole_m^i(T)$ can be grouped.

The above correctness proofs, of the elimination and grouping, together with the correctness proof of the CNT algorithm gives the correctness of the CNT-RS algorithm.

3 Space complexity

An important aspect of the dispatch algorithm is how much memory is used by it. In this section the two algorithm's space requirements, expressed in the number of poles, is presented and a comparison between them is made. This is followed by an example showing that you can get an uncompressed case even with the CNT-rs algorithm.

3.1 Unit of Measurement

The space complexity of the algorithms is dependent on the inheritance hierarchy and the apportionment of arity of the methods. In this case we will only look at the size of the dispatch tables for one method at a time, because this is what is needed to be able to compare the two algorithms. The space requirement will be looked at as the number of table cells needed.

3.2 CNT

The space requirement measured in table cells needed and expressed in the number of poles. In the first dimension you need $\left|Pole_m^1\right|$ table cells. In the second dimension $\left|Pole_m^1\right|\left|Pole_m^2\right|$ table cells is needed and so on. So the total number of table cells needed is as follows.

$$\text{Number of table cells for method } m = \sum_{i=1}^{n}\prod_{k=1}^{i}\left|Pole_m^k\right|$$

3.3 CNT-RS

The space requirement measured in table cells needed and expressed in the number of poles. In the first dimension $\left|MPole_m^1\right|$ table cells is

needed. In the second dimension $\left|MPole_m^1\right\|Pole_m^2\right|$ table cells is needed. In the third dimension $\left|MPole_m^2\right\|Pole_m^3\right|$ table cells is needed and so on.

Number of table cells for method $m = \left|MPole_m^1\right| + \sum_{i=2}^{n}\left|MPole_m^{i-1}\right\|Pole_m^i\right|$

3.4 Comparison of CNT and CNT-RS

The number of multipoles at dimension i is the same or less than the numbers of types in the new type hierarchy. The new type hierarchy created at dimension i has $\left|MPole_m^{i-1}\right\|Pole_m^i\right|$ new types.

When i = 1 we have $\left|MPole_m^1\right| = \left|Pole_m^1\right|$.

Assume when i = j that $\left|MPole_m^j\right| \le \prod_{k=1}^{j}\left|Pole_m^k\right|$ holds.

When i= j+1:

$\left|MPole_m^{j+1}\right| \le \left|Pole_m^{j+1}\right\|MPole_m^j\right| \le \left|Pole_m^{j+1}\right|\prod_{k=1}^{j}\left|Pole_m^k\right| = \prod_{k=1}^{j+1}\left|Pole_m^k\right|$. The induction principle gives that $\left|MPole_m^i\right| \le \prod_{k=1}^{i}\left|Pole_m^k\right|$ holds for $i \ge 1$.

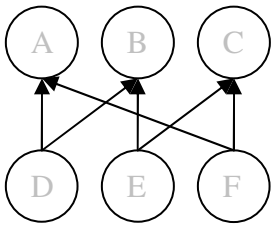
From this we get that $\left|MPole_m^1\right| + \sum_{i=2}^{n}\left|MPole_m^{i-1}\right\|Pole_m^i\right| \le \sum_{i=1}^{n}\prod_{k=1}^{i}\left|Pole_m^k\right|$.

And in words that means that CNT-RS is at least as good, space efficient, as CNT.

3.5 A worst case example of CNT-RS

From the above comparison of CNT and CNT-RS we know that CNT-RS is at least as good as CNT, but CNT can give an uncompressed table without having to have a method for every possible combination of arguments. Here we will present a worst case, uncompressed table,

example for the CNT-RS algorithm. We get an uncompressed table if all types/multitypes in the hierarchies are poles/multipoles.



$m_1(A, A, A)$
$m_2(B, B, B)$
$m_3(C, C, C)$
$m_4(A, B, A)$
$m_5(A, C, A)$
$m_6(B, A, A)$
$m_7(B, C, A)$
$m_8(C, A, A)$
$m_9(C, B, A)$

Figure 16 Example that gives an uncompressed table

From this we compute in the same way as above and get that A, B and C are primary poles and D, E and F are secondary poles for all three argument positions. That is to say that the pole hierarchies are the same as the type hierarchy. To get the pole hierarchies like this we would only have needed m_1, m_2 and m_3. So to get an uncompressed table with CNT we would only have needed these first three methods, but for CNT-RS we need at least all the methods stated above. To see why they are needed let's create the second multitype hierarchy and see how we can make all the multitypes into multipoles.

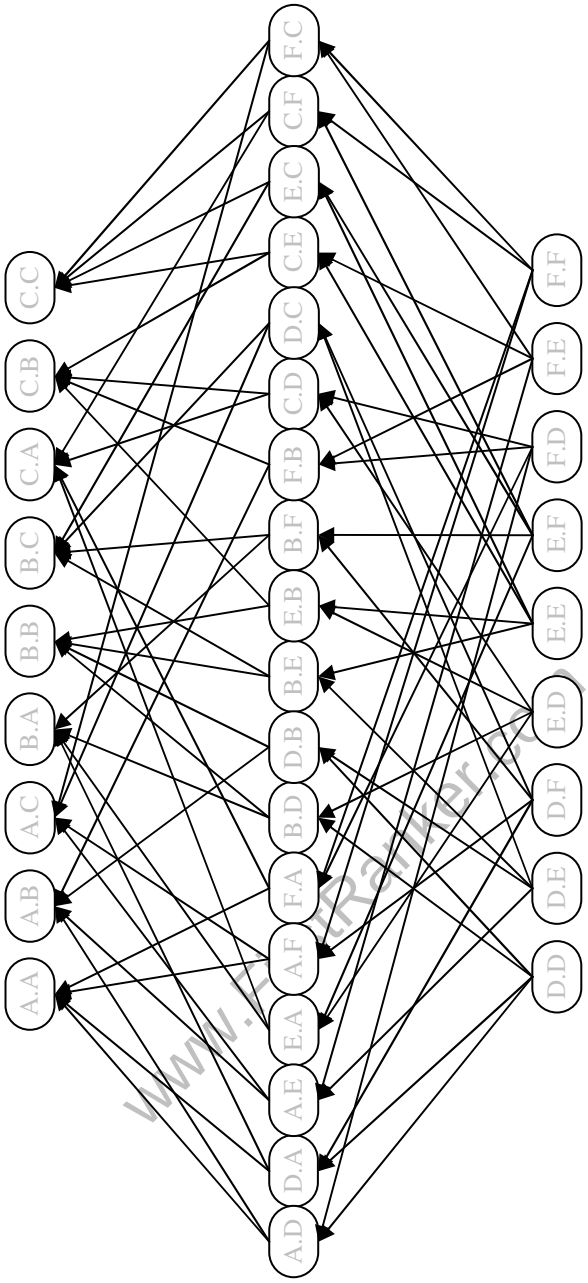


Figure 17 The second multitype hierarchy of the worst case example

As we can see from the multitype hierarchy the top row has no parents and has to be primary multipoles to avoid being eliminated. If all the multitypes in the top row are multipoles we see that all multitypes in the second row are secondary multipoles as they have two parents associated with different unrelated multitypes. For the third row it's the same thing but with four parents so all the multitypes in the third row also are multipoles and we now know that all multitypes are multipoles.

As the third and last multipoles hierarchy doesn't affect the size of the dispatch table, and that is all we are interested in in this case, we don't need to bother about it.

We can now see why we needed the above nine methods to get an uncompressed table. This example shows us that only nine of 216 possible methods are needed to get an uncompressed dispatch table if the type hierarchy and methods are very unfavourable, but it still better than the CNT's three.

4 Future work

Some kind of amortised, "on average", value for the space complexity should be calculated. This is interesting because we know that the CNT-RS algorithm gives at least as small tables as the CNT, but we don't know what happens "on average". The space complexity should be lower on average, but is it? On average, do the two algorithms grow in the same way? And what is this on average, how will a real life system written with multiple dispatch look like on average?

There are some improvements possible on the CNT-RS algorithm. For example the missed grouping described above. How much better can it be and at what cost?

A "real life" test on the de facto benchmark systems, Cecil [2] and Dylan[1], should be performed. This should be done to verify the results in this report and to give concrete numbers to compare to other multiple dispatch algorithms.

A comparison of space requirement between the CNT-RS algorithm and the MRD algorithm. MRD does the same elimination and grouping as CNT and save some more space by the row displacement. How does the CNT-RS and MRD compare? How does the use of row-shifting or row-matching in MRD affect the difference.

Does the ordering of the combination of arguments give any significant difference in space requirement? Is there a better way to combine the poles than first and second, first, second and third…?

5 Conclusion

For the development of the programming language Scream, within the Software renaissance project, a good multiple dispatch algorithm was needed and the CNT-RS algorithm was developed. The CNT-RS algorithm is an extension of the CNT algorithm, an algorithm often referred to in the area of multiple dispatch.

The added elimination and grouping are proven to be correct and as the CNT algorithm already are proven to be correct the CNT-RS algorithm is correct.

The CNT-RS algorithm is more space efficient than the CNT algorithm. Though some more work should be done to, especially see how the algorithm performs in real life, and compares to other multiple dispatch algorithms.

References

1. Apple Computer: Dylan reference manual, Apple Computer, Cupertino, Calif. 1995
2. C. Chambers: Object-oriented multimethods in Cecil, ECOOP Conference Proceedings, Springer-Verlag, 1992
3. E. Dujardin, E. Amiel, E. Simon: Fast Algorithms for Compressed Multimethod Dispatch Table Generation, ACM Transactions on Programming Languages and Systems, p116-165, January 1998, Volume 20, Number 1.
4. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley, 16th Printing, December 1998.
5. W. Holst, D. Szafron, C. Pang, Y, Leontiev: Multi-Method Dispatch Using Single–Receiver Projections, Technical Report TR-98-03, University of Alberta, Edmonton, Canada, 1998.
6. B. Hägglund: Design av ett anpassningsbart programspråk, LiTH-IDA-Ex-02/49, Linköpings universitet, 2002-04-23
7. C. Pang, W. Holst, Y. Leontiev, D. Szafron: Multi-Method Dispatch Using Multiple Row Displacement, ECOOP'99 Proceedings of the 13th European Conference on Object-Oriented Programming, p304-328.
8. Softren, www.softren.org, 2004-02-17

LINKÖPINGS UNIVERSITET

Avdelning, Institution
Division, Department

Institutionen för datavetenskap
581 83 LINKÖPING

Datum
Date
2004-02-17

Språk
Language
Svenska/Swedish
X
Engelska/English

Rapporttyp
Report category
Licentiatavhandling
X Examensarbete
C-uppsats
D-uppsats
Övrig rapport
____

ISBN

ISRN LITH-IDA-EX--04/018--SE

Serietitel och serienummer **ISSN**
Title of series, numbering

URL för elektronisk version
http://www.ep.liu.se/exjobb/ida/2004/dd-d/018/

Titel Analysis of a multiple dispatch algorithm
Title

Författare Johannes Holmberg
Author

Sammanfattning
Abstract
The development of the new programming language Scream, within the project Software Renaissance, led to the need of a good multiple dispatch algorithm. A multiple dispatch algorithm, called Compressed n-dimensional table with row sharing; CNT-RS, was developed from the algorithm Compressed n-dimensional table, CNT. The purpose of CNT-RS was to create a more efficient algorithm. This report is the result of the work to analyse the CNT-RS algorithm. In this report the domain of multiple dispatch, the multiple dispatch algorithm CNT and the new extended algorithm CNT-RS are presented. The correctness of CNT- RS algorithm is shown and it's proven that the CNT-RS algorithm is at least as good as the CNT algorithm, in regards to space complexity of the dispatch structure.

Nyckelord
Keyword
dispatch, multiple dispatch, dispatch table, pole, multipole, influence, type hierarchy, pole hierarchy, multipole hierarchy