Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 257–264.

Comparative analysis of refactoring and code optimization*

Róbert Kitlei, Gergely Dévai, Ádám Balogh, Zoltán Csörnyei

Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University, Budapest

Abstract

There are two basic types of program transformations, the translations, where the source and target languages are different, and rephrasings, where these languages are same. Typical example for translations is a compiler, and one example for rephrasings is the code optimization of compilers and the second is refactoring. Both code optimizaton and refactoring are meaning-preserving transformations.

Refactoring restructures the program to improve its design [3, 4, 6] and the optimization improves its run-time and space performance [1, 2, 5].

Code optimization differs from refactoring in this respect, the goal of optimization is to improve the speed and space usage and the goal of refactoring has a variety of reasons, such as extensibility or flexibility. Despite the differences, the techniques used for refactoring are similar or identical to the techniques used to optimize the code.

In this paper these similarities and dissimilarities are investigated. We study how to apply methods of code optimization in refactoring techniques to make it an even more powerful tool to develop safe and effective programs.

1. Motivation

Refactoring is a technique that has been in use for long in practice, however, its automation and formalisation has not been in the centre of attention until recently. It changes the source code of a program in order to improve its quality. This in turn enhances the whole software product, as it makes further development easier.

Optimisation, in contrast, has been an interest of development since the beginnings of computing. In the beginning, computers had limited resources, therefore

*Supported by GVOP-3.3.3-2004-07-0005/3.0 ELTE IKKK.

intricate techniques were needed to make sparing use of them. Also, the development of programming languages has brought about many constructs that would be costly to implement done the straightforward way. Because of all this effort, many sophisticated techniques have evolved in this area.

These two technologies have existed independently of each other for some time for diverse languages (of different programming paradigms, even). In our paper we show that they possess many aspects in common. Researchers of both may benefit from the more developed counterparts.

2. Introduction of the topics

2.1. Refactoring

Refactoring is done when the software developer judges the source code inadequate in some way. As the name suggests, the goal is to find a better way to factor the task – in other words, to change the structure of the program so that it reflects that of the problem more closely. Most of the time the changes required, even a relatively small change as the renaming of a variable, effect a large part of the code. A common criterion is that the program semantics have to remain the same, and a local change could influence other structures as well.

Doing refactoring is tedious and error prone because of this non-locality. Users in the past would "cut and paste" the changed parts to their new locations, but doing this they frequently missed an occurrence or changed part of the code in error. Parametrised refactorings are even less feasible by hand because of the high level of attention needed. If the language in question has more complex structures, like the dynamic bindings of object oriented languages, refactoring becomes even more difficult.

The state of the art way of refactoring is to delegate all the house-keeping to a tool that has been specifically designed for this task. The programmer interacts with the tool only giving the essential information about the refactoring, as the new name of the variable, and the tool either performs all the necessary changes, or returns an error message if it is impossible to perform the refactoring, for example when the name is already assigned to another variable.

2.1.1. An example

```
int f() {
  int j = 0;
  for ( int i = 0; i < 100; ++i )     j += i;
  return j;
}
```

This example shows a C code fragment that contains names that do not indicate their purpose clearly. With reformatting, one can obtain the following code.

```
int calculateSumUpTo( int upperLimit ) {
  int result = 0;
  for ( int i = 0; i < upperLimit; ++i )    result += i;
  return result;
}
```

The goal of this function is understood better at the place of calling simply because of its name.

2.2. Compiler optimisation

The goal of optimisation is to improve the performance of programs, at least in the general case. It might decrease running time, memory usage or the length of generated code; usually there is a trade-off between these goals. Legibility of the output is not a requirement, for the user rarely if ever encounters the produced code.

Compiler optimisation is performed by the compiler during compilation time without any interaction from the user (the level of optimisation – the set of optimisation techniques to be applied – might be set before the start of compilation). An important difference from refactoring is that because of the full automation, the transformations are always performed in one way, while refactoring has the flexibility to use both ways if the transformation is eligible.

Optimisation, like refactoring, should not change the observable behaviour of the code. This depends on the abstraction level of the observer, for example additional criteria need to be added when talking about parallel programs, therefore it should be agreed upon beforehand, what is malleable and what is to be kept. Some compilers offer various levels of optimisation that are suitable for different purposes. Usually the techniques that give better performance are apt to violate more requirements.

2.2.1. An example

```
int f()
{
  int j = 0;
  for ( int i = 0; i =< 100; ++i )    j += i;
  return j;
}
```

This is the same piece of code that we had looked at before. From the performance point of view it is inefficient, because it calculates a sum that can be expressed directly. An optimising compiler can produce code from this source equivalent to the following.

```
int f() {
  return 5050;
}
```

Note that while it is never unavoidable to perform refactoring, optimisation may be necessary.

```
fac 0 = 1
fac n = n * fac (n-1)
```

This example shows a recursive function in a functional programming language like Haskell or Clean, that would consume memory proportional to the number of iterations. With an optimisation that is called tail recursion elimination, this can be reduced to a constant.

3. Some basic techniques

3.1. Techiques common for refactoring

3.1.1. Renaming

```
i := geticlmt()
```

This assignment is not literate.

```
invoiceableCreditLimit := getInvoiceableCreditLimit()
```

After renaming the variable and the function the assignment makes more sense. This sort of transformation is used often in refactoring practice, but it can not apply to optimisation: for the latter the names of symbols are irrelevant except for their use to tell the symbols apart.

3.1.2. Restructuring

Refactoring has a broad range of operations that move variables, functions, methods, blocks of code, or change class hierarchies. Using these appropriately can greatly improve the coupling of the code.

However, these types of operations are not applicable in code optimisation because of the need for human interactions. Automatic recognition of ideal structures would only theoretically be possible due to it requiring excessive amounts of resources.

3.2. Techniques that are used in both areas

3.2.1. Expression simplification

Expression simplification is one kind of local optimisation. These are transformations that affect a limited range of code, with no branching.

```
a := b + 1 + c + 3 + 4
```

By reordering and fusing the constants we get the following.

```
a := b + c + 8
```

Expression simplification can be done if the operation is commutative, and no side effects are present.

3.2.2. Common subexpression elimination

Common subexpression elimination is another kind of local optimisation.

```
j := 2 * d
i := j
k := -i
j := i + 3
d := d + 2
j := j + d
```

This transformation reorders assignments so that they can be performed faster. Also, it may construct an alternative way of calculating the needed expressions.

```
d := 2 * d
i := d
k := -i
j := 2 * d + 3
```

3.2.3. Loop optimisation

```
do i = 2 to 8 by 2
  a[ i, j ] := 0
end
```

This code fragment clears selected elements of an array.

```
a[ 2, j ] := 0
a[ 4, j ] := 0
a[ 6, j ] := 0
a[ 8, j ] := 0
```

If loop unrolling is applied, the loop overhead is gone at the cost of increased code size. When refactoring, transformation int the opposite direction might be more appealing. It is possible to only partially unroll the loop.

3.2.4. Code hoisting

```
do i = 1 to 100
  b := 2
  a := a + f( b )
end
```

This example shows a loop with a body containing an invariant that can be removed outside of it.

```
b := 2
do i = 1 to 100
  a := a + f( b )
end
```

This way the expression b := 2 is evaluated only once, and the structure of the code reflects the invariance of the value of b in the loop more explicitely.

3.2.5. Code inlining and extraction

```
int f(int i) { return min(g(i) + h(i)); }
...
x = 3;
x += f(10);
```

The overhead of calling the function f and passing the parameter can be spared if we inline the call.

```
x = 3;
x += min(g(10) + h(10));
```

This is a transformation that can be useful for the purposes of refactoring both ways. The programmer has to decide whether the function represents an added level of abstraction or the body is inherent to the expression. Another benefit of extracting parts of code is the further possibility of generalising the code as needed, for example like this.

```
int f(int i, int j) { return min(g(i) + h(j)); }
```

3.3. Our current work

We are currently implementing refactorings for Erlang, a parallel functional programming language with strict evaluation, single assignment, and dynamic typing designed by Ericsson. Several of the above mentioned transformations are completed or planned. Our efforts are currently directed toward a transformation called merge subexpression duplicates, that works the following way.

```
foo(A, B) ->
  peer ! {note, A + B},
  A + B.
```

The function foo in this code fragment sends the peer a note that contains the sum of its parameters, then returns the same value. This can be made more efficient and readable if the sum is calculated only once.

```
foo(A, B) ->
  C = A + B,
  peer ! {note, C},
  C.
```

A new variable (whose name is given by the user) is placed directly before the first occurrence of the expression, and the value of the expression is bound to it. All occurrences of the expression are replaced by this variable thereafter.

This technique is closely related to common subexpression elimination. Theoretically it would be possible to perform it as an optimisation that merges all possible subexpressions, but it would require too many resources. As refactoring the user chooses only one expression, for which calculations are computationally feasible.

4. Conclusion

In this paper we have given a brief introduction to refactoring and compiler optimisation. We have shown that these methodologies share many traits and techniques that are related to one another, and that developers and programmers of each area may benefit from reviewing techniques from the other. As optimisation is more developed, probably refactoring shall receive more from this, as is the case in our example.

We have given a brief overview of our ongoing project. It concerns a refactoring that is a close relative to an established optimisation technique.

References

[1] Allen, R., Kennedy, K., Optimizing compiler for modern architectures, *Morgan Kaufman*, (2002).

[2] Csörnyei, Z., Fordítóprogramok, *Typotex*, (2006).

[3] Diviánszky, P., Szabó-Nacsa, R., Horváth, Z., A Framework for Refactoring Clean Programs, *Proceedings of the 6th International Conference on Applied Informatics*, Eger, Hungary, (January 27–31, 2004), Vol. I., 129–136.

[4] Fowler, M., Refactoring: Improving the Design of Existing Code, *Addison-Wesley*, (1999).

[5] Grune, D., Bal, H. E., Jacobs, C. J. H., Langendoen, K. G., Modern Compiler Design, *John Wiley & Sons*, (2000).

[6] de Jonge, M., Visser, E., Visser, J., XT: a bundle of program transformation tools, *Electronic Notes in Theoretical Computer Science, 44* (2001), No. 2.