

Exploring the Design of Compiler Feedback

THIBAUT RAFFAILLAC



**KTH Computer Science
and Communication**

Master of Science Thesis
Stockholm, Sweden 2012

Exploring the Design of Compiler Feedback

T H I B A U L T R A F F A I L L A C

DH224X, Master's Thesis in Human-Computer Interaction (30 ECTS credits)
Degree Progr. in Computer Science and Engineering 270 credits
Royal Institute of Technology year 2012
Supervisor at CSC was Ylva Fernaeus
Examiner was Henrik Artman

TRITA-CSC-E 2012:084
ISRN-KTH/CSC/E--12/084--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

Exploring the design of compiler feedback

Abstract

Nowadays, programmers willing to start optimising their code must undergo a lengthy interaction with dedicated profiling tools. This Degree Thesis proposes as an alternative to make compilers generate feedback messages aimed at explaining how they understand the code, and how it could be improved. The study aims at foreseeing the technical integration of feedback notifications in modern compilers, as well as sketching how Integrated Development Environments (IDE) would display them.

A first comparison of three related works enables the core differentiators to be highlighted: letting the compiler inform where code is actually fine and does not need any refinement, displaying the notifications along the relevant source lines rather than in a separate interface, insisting on the absence of artificial intelligence, and introducing a filter heuristic to take into account the less significant messages. Then, a preparatory user study is carried to observe different programmers and poll their receptiveness to a compiler feedback. The findings relate the usefulness of optimisations' suggestions to fit where users lack expert knowledge, the existence of dormant interrogations calling for serendipitous information retrieval, and the mistakes inherent to Message of the Day windows which should be avoided.

Three prototypes are designed to embody three different approaches, using Web tools to provide a close appearance to code editors along with decent interactivity. With the help of a new user study with the prototypes, a final set of refinements is discussed so as to shape a coherent result and differentiate it further: users can create and share sets of feedback messages to supplement the ones included in their compiler, a list of rules is provided to help designers compose the messages, an emphasis is laid on transparency to help exhibit the absence of artificial intelligence, and the heuristic used to sort and filter the displayable notifications is sketched.

Utforskning av kompilatorfeedbacksdesign

Sammanfattning

Programmerare som vill optimera sin kod måste normalt genomgå en ganska omständlig process med hjälp av ett dedikerat profileringsverktyg. Detta examensarbete diskuterar olika alternativ där kompilatorn mer direkt genererar återkoppling på hur den tolkat koden, och hur den kan förbättras. Studien syftar till att ge ökad insikt i utvecklingen mot teknisk integrering av feedback-tillämpningar i moderna kompilatorer, samt att skissa på hur det skulle kunna se ut om de visades i integrerade utvecklingsmiljöer (IDE).

En första jämförelse av tre relaterade arbeten ledde fram till några utmärkande egenskaper att arbeta för, kompilatorernas nuvarande sätt att informera om koden är redan bra och behöver inte någon förfining, att visa notifieringar längs den berörda källraden stället i ett separat gränssnitt, att undvika lösningar som bygger på artificiell intelligens, och att införa ett filter som heuristisk tar hänsyn till mindre viktiga meddelanden. Därefter tillsattes en förstudie där inställningen till kompilatorfeedback undersöktes bland en grupp programmerare. Resultaten relaterade nyttan med förslag på optimeringar när användarna saknar expertkunskap, slumpartad informationssökning, och problem som t ex att ”dagens meddelande” bör undvikas.

Tre prototyper utformades för att förkroppsliga tre olika metoder för hur detta skulle kunna ta form, dessa presenteras online för att ge ett nära utseende av verkliga kodbehandlare och erbjuda enkel interaktivitet. Efter en slutlig användarstudie med dessa prototyper reviderades uppsättningen med finesser och förslag med en ytterligare specialisering, dvs att låta användare skapa och dela lämpliga feedback-meddelanden som kompletterar de som ingår i kompilatorn själv, en förteckning över regler för att hjälpa konstruktörer skriva meddelanden, med en betoning på öppenhet för att uppvisa avsaknad av artificiell intelligens, samt utarbetning av tumregler för att sortera och filtrera visningsbara notifikationer.

Table of Contents

1. Introduction.....	1
1.1. Problem definition.....	2
1.2. Challenges.....	3
1.3. Method.....	4
2. Related work.....	6
2.1. Research methodology.....	6
2.2. Study of three similar efforts.....	6
a) VISTA, the vpo Interactive System for Tuning Applications.....	7
b) EAVE, the Expert Adviser for Vectorization.....	8
c) Matlab Code Analyzer.....	9
2.3. Rationale and differentiators.....	9
a) The compiler queries the programmer.....	9
b) No separate interface.....	11
c) Cooperation instead of assistance.....	11
d) The filtered notifications.....	12
3. Preparatory study.....	13
3.1. The interview plan.....	13
3.2. Conducting the interviews.....	15
a) Problem solving.....	16
b) Subsequent findings.....	17
4. Three prototypes.....	19
4.1. First prototype: a stripped version for GCC.....	19
4.2. Second prototype: a communicative compiler.....	25
4.3. Third prototype: a far-fetched alternative.....	28
5. User study with the three prototypes.....	30
5.1. A new round of interviews.....	30
5.2. Findings and suggestions for future work.....	31
a) The personae.....	31
b) A few rules for composing the messages.....	32
c) Transparency is crucial.....	32
d) A proposed formula for the sorting and filtering of messages.....	33
6. Discussion and concluding words.....	34
6.1. Limitations.....	34
6.2. Personal conclusions.....	35
References.....	36
Appendix A.....	37
Appendix B.....	40

1. Introduction

Compiling a program nowadays is simple. Once the source code itself is written, a single action is needed to turn it to an executable file. With an Integrated Development Environment (IDE) such as Eclipse or Microsoft Visual Studio, it is synonym with clicking on a “Build” button. With a command-line compiler such as the GNU Compiler Collection (GCC), it is computed with a single command. Past the errors and warnings, as soon as a working executable is output the compiler has fulfilled its task; it will not provide any more interaction.

When it comes to optimisation, however, the procedure becomes trickier. By setting the proper options and command-line flags, it is normally handled transparently by the compiler, yet in practice this support is irregular (see Aho, Lam, Sethi, & Ullman, 2006, for a technical overview of modern compilers). While instruction scheduling and register allocation are decently achieved nowadays¹, improvements such as making parallel loops or exploit the locality of memory accesses *will* require tuning specific options, or the source code itself².

On the other hand, a significant knowledge gap separates the programmer from the compiler. For the former, the language syntax³, the diversity of architectures and systems, and the basic skills required for a software engineer (Kreeger, 2009; Lethbridge, 2000), are potentially overwhelming. For the latter, as quoted from Bose (1988), *the compiler is not designed to fully “understand” the high-level, algorithmic intentions expressed by the user in his (or her) source code.*

As a consequence, it is believed that users do not obtain the performance and security they should expect from their programs. During this Degree Project I investigated the design and integration of feedback messages from the compiler, to leverage opportunities of tuning and improve the user's mastery in software programming.

The data collected here is based on literature and user studies as well as my own experience as a C/C++ programmer with a passion for performance. The focus will therefore be on C++, as this general-purpose language is widely used in industry nowadays. Furthermore, most of the examples in this work focus on improving performance, though I have put many efforts in diversifying them.

1 With the example of GCC, see <http://gcc.gnu.org/wiki/InstructionScheduling> and <http://gcc.gnu.org/wiki/RegisterAllocation> (both accessed 03.09.2012).

2 See the example of GCC at <http://gcc.gnu.org/onlinedocs/libgomp/Enabling-OpenMP.html>, and for Intel at <http://software.intel.com/articles/automatic-parallelization-with-intel-compilers/> (both accessed 03.09.2012).

3 Refer to the C++ specification, for example (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372, accessed 03.09.2012).

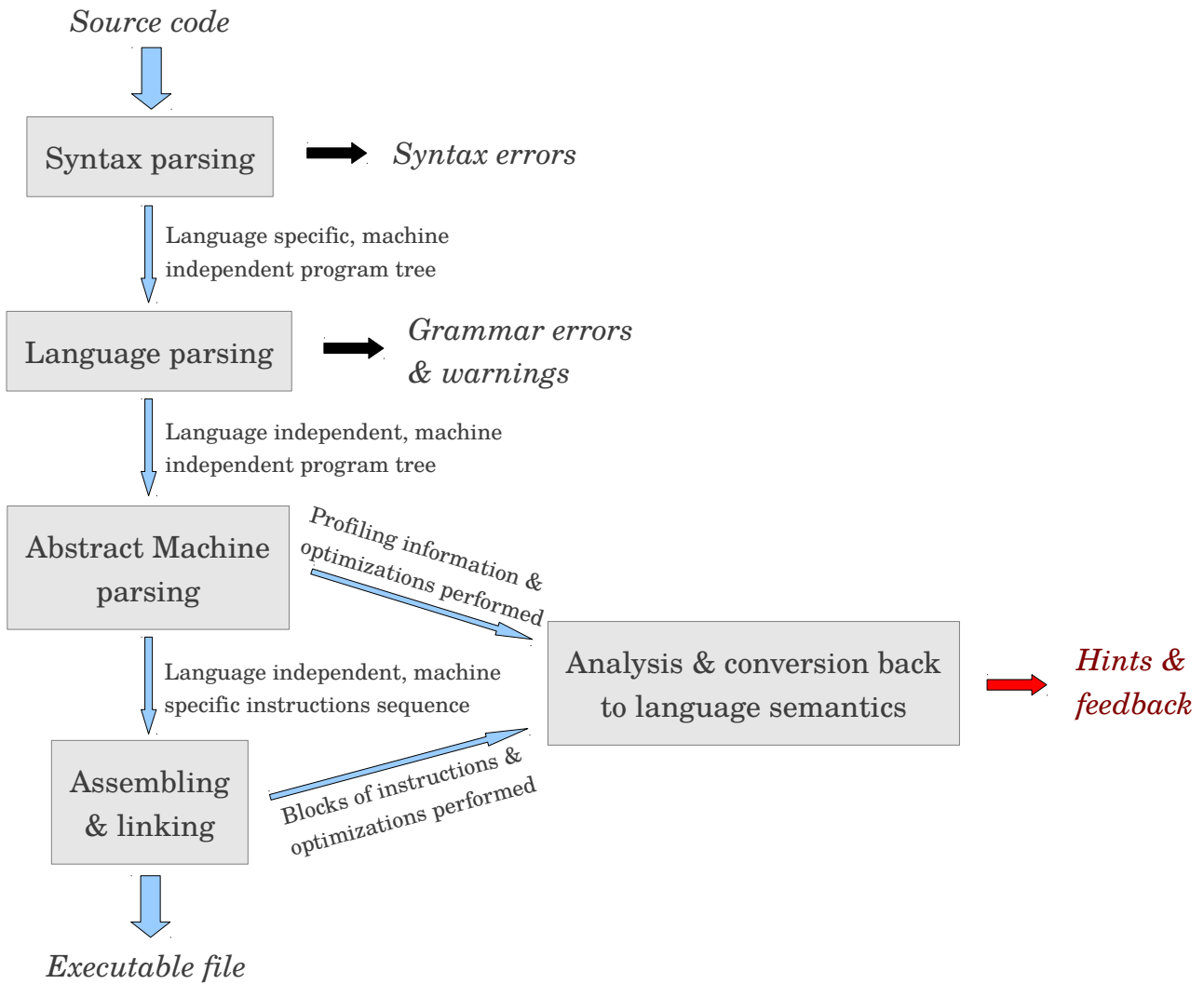


Figure 1: Inclusion of a feedback in the stages of compilation

1.1.Problem definition

The compiler should extend interaction *after* the creation of a working executable, to suggest opportunities of improvement, for example. Software such as Intel Vtune Amplifier, SmartBear Aqtime Pro, or Microsoft Visual Studio Analyzer, can typically provide this functionality. However, they are rather a collection of tools, and require dedicated learning, yet I do not intend to *dig* the knowledge gap.

A proposed solution in this Thesis is to make the compiler output *feedback messages*, pretty similar to the existing warnings and errors (Figure 1). Fundamentally, they would provide information on the compiler's operation while parsing the source code, on how it was “understood”. In addition to suggesting code improvements, they could assure that a hand-coded optimisation is unnecessary when it is transparently done by the compiler, recommend the use of a little-known standard feature, or introduce and recommend a compiler-specific feature. Refer to the prototypes further for more

examples.

Beyond feedback, with a little more work the compiler could be able to directly query the programmer, to suggest local tunings when the language semantics show their limits. Think about the problem of specifying the underlying search tree structure of a `set` object. Since the C++ standard library does not offer this choice, users must cope with the default implementation shipped with their compiler. If the latter provides several implementations though, it could probably be specified through *pragmas* (the compiler-specific preprocessing instruction in C/C++), but again this would require learning the particular syntax. An alternative here would be to generate a query with radio buttons for the programmer, and automatically insert the correct corresponding `#pragma` call.

Furthermore, thanks to the introduction of feedback, conditional compilation features such as preprocessing and generics could be removed from programming languages. Nowadays, compilers are capable of evaluating certain run-time expressions at compile-time. With lack of information, however, users still resort to conditional compilation to enforce the evaluation of expressions during the compilation phase. Now, if a user is aware of *which conditions* trigger the former behaviour, it is no longer necessary to separate run-time from compile-time semantics, leading to a lighter language.

My work along this Thesis was thus dedicated to solving the following problem:

- How can compiler feedback be designed and integrated in modern compilers?

1.2. Challenges

Many potential issues could be foreseen ahead of this work, this section lists them and refers to the respective chapters where they are handled.

The biggest difficulty is certainly to properly identify the context, that is to output messages which actually *interest* the users. Some might be concerned about performance, some about security, some about memory usage, etc. The risk is that one faulty message makes the user disable the whole functionality, as with the *Message of the day* window (see the Preparatory study). To ensure a satisfactory level for notifications, I rely on their technicality and transparency, as discussed in the preparatory interviews and the prototypes' tests.

On the software side, relating a low-level transformation to the original source code is a technically challenging task (shown as “conversion back to language semantics” in Figure 1), also the messages will certainly depend on the architecture and system targeted. To satisfy this exigence of accuracy, a syntax for *trigger* conditions is

discussed starting with the first prototype.

Besides, a careless solution can be faced with numerous issues. Indeed, allowing the compiler to suggest improvements, select the most valuable ones, or even query the programmer, could be synonym with artificial intelligence. This is out of question here, both because such a difficult solution would never be accepted among compiler developers, and because it is contradictory with the requirement of transparency introduced above. Moreover, the notifications are to be triggered after user input (running the compilation), to sidestep the issues bound to random occurrences, as highlighted in Carroll (1988). Indeed, it will avoid the frustration of receiving a suggestion precisely after having figured it out, and the user will not be distracted by the expectation of a new message.

As the reader can expect, for a reasonable project the amount of notifications generated will be huge. Being of lower importance compared to warnings and errors, they do not have to be displayed all at once each time a successful compilation is performed. Instead, a formula is proposed to sort the messages and filter in the most important ones, as discussed after the user study.

As a last point, the neutrality of messages is also a potential issue. It concerns how to guarantee that the assumptions and figures argued are accurate. To tackle this challenge, a source will systematically be cited, providing a hypertext link wherever possible. In addition, the author (or organisation) might be explicitly bound to each message, as discussed in the end of this thesis. Nevertheless, a subjective point-of-view is somehow desirable, as it will reflect the advices (and perhaps the personality) of the interface designer.

1.3. Method

The structure of the thesis was based on Saffer (2009) proposed methodology, to avoid overlooking such crucial steps as the search for differentiators, and the analysis of the data gathered from the preparatory interviews. I chose to focus on User-Centered Design, to extensively query different programmers so as to balance my biased passion for optimising. My main concern was indeed that users would not be interested in refining their code, and that they could disable the compiler feedback for the lack of short-term usefulness.

The thesis starts with a comparison of three previous similar works. The problem of improving the accessibility of profiling and optimising is not new, one should thus ensure that their findings and pitfalls are acknowledged, and clearly identifying how this work will take a different approach.

The preparatory interviews which followed were intended as an occasion to observe

different users program, and the difficulties for which they would appreciate help. I was not looking for an open solution though. Instead, the goal was already set – introducing a compiler feedback – and I was willing to shape it as best to satisfy and help the users.

Though initially unplanned, I chose to apply for a Google Summer of Code during this Thesis work. This annual event is an opportunity for students to contribute to an open source project, while being paid. It is mentioned in this report since it greatly helped the definition of the first prototype, both technically and for its design. This was followed by two other prototypes, supplementing with alternate solutions to introduce compiler feedback.

A series of tests was then carried during a new user study, so as to gather helpful comments for further iterations. The prototypes being non functional though, I did not deem useful to actually refine them, and focused on wording the last key differentiators.

It should be noted at last that I used the results from previous personal studies – namely the enumeration of the design aspects of C++, and a list of ideas for feedback messages gathered during several months of programming – as a basis for the enumeration of suggested messages.

2. Related work

This chapter covers the study of previous similar attempts and the resulting highlighting of improvements brought by a compiler feedback, in order to ensure its competitiveness.

2.1. Research methodology

The gathering of reference papers received a careful attention during this Degree Project. It mainly consisted in browsing the archives of a few Special Interest Groups (SIG) on the online portal of the Association for Computing Machinery, plus a thorough keyword search on Google Scholar. Some papers were also found by following cross references. The scope being potentially broad, various interest groups were targeted:

- The group on Computer-Human Interaction (10 years of Transactions on Computer-Human Interaction)
- The groups on Computers and Society, Computer Science Education, and Information Technology Education (5 years of SIGCAS archives, 5 years of SIGCSE Bulletin, and 10 years of SIGITE Newsletter)
- The group on Algorithms and Computation Theory (5 years of SIGACT News)

For the sake of completeness, here are the keywords which were used to search on Google Scholar: *static profiling [visualisation], optimisations visualisation, compiler [interface], [serendipitous] programming learning, embodied interface, programmer improvement, programming best practices, recommender system /agent, [interactive] program improvement.*

2.2. Study of three similar efforts

The problem of improving the interface of compilers has existed for many decades now (see Bose, 1988, presented further), and various solutions have been proposed. Let us present three related approaches, and distinguish how each could be improved.

a) VISTA, the vpo Interactive System for Tuning Applications

VISTA (Zhao et al., 2002) is a system targeting embedded applications, for which assembly optimisations are often necessary to ensure speed, low power consumption and decent size of the executable. Along with a graphical visualisation of the Register Transfer Language (RTL), the user can reorder the code improvement phases, manually specify code transformations and profile the different compiled versions for comparison purposes (Figure 2).

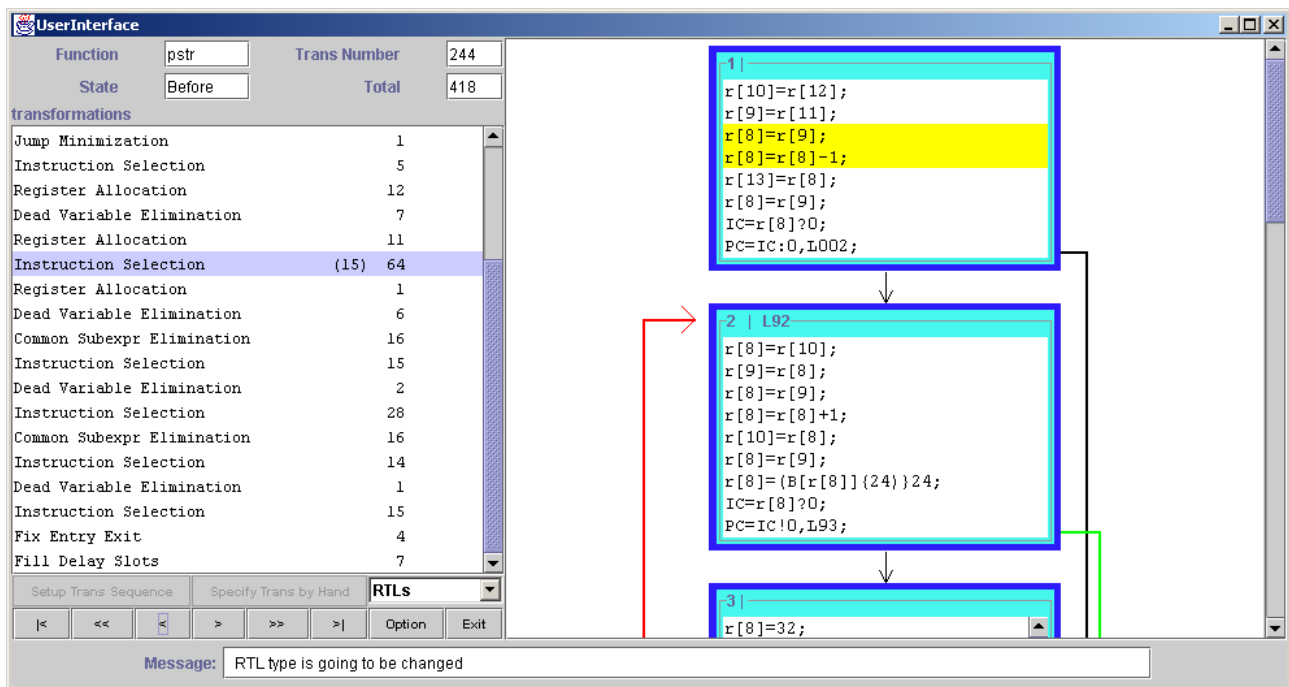


Figure 2: Interface of VISTA, as presented in Zhao et al., 2002

However, the interface should give clues as to which particular orders of transformations usually provide descent results. To actually improve the program's performance, the user has to undergo a lengthy trial and error, testing all combinations of passes.

Additionally, the designers are asserting that the users are already familiar with the optimisation techniques involved. While as embedded applications' programmers they should certainly own some knowledge, it is extremely unlikely that they are familiar with all of the presented techniques. It is too easy to assume that the programmer has access to a book, a course or any relevant reference document, as is too easy to expect that he/she will actually have the will to browse it.

As for the RTL, it is as expanded a translation of the source code as the resulting assembly code is. Understanding it and verifying the expected interpretation is finally equivalent in time to writing the assembly itself. The interface should instead be relating to the original source code.

b) EAVE, the Expert Adviser for Vectorization

EAVE (Bose, 1988) is a helper agent designed for programming in Fortran on the IBM 3090 VF computer. Its purpose is to help programmers achieve near-peak performance by advising the proper loop constructions which the compiler will vectorise or parallelise (Figure 3).

What is the DO loop to be analyzed?	PF1 Help
(If you would like to enter the loop line by line, press PF6)	PF2 Review
(Enter DO LOOP, line by line)	PF3 End
	PF4 What
	PF5 Question
	PF6 Unknown
	PF7 Up
	PF8 Down
	PF9 Tab
	PF10 How
	PF11 Why
	PF12 Command


```
dimension a(100, 100), b(100, 100), c(100, 100)_____
do 10 i = 1, 100_____
do 20 j = 1, 100_____
c(i, j) = 0_____
do 30 k = 1, 100_____

c(i, j) = c(i, j) + b(k, j) * a(i, k)_____
30 continue_____
20 continue_____
10 continue_____
.
```

Figure 3: Excerpt of EAVE's interface as presented in Bose, 1988

However, the interaction with the agent is lengthy. One has to provide the source code, point the loop to be analysed, then run the analysis. Moreover, this obligation to *request* feedback prevents opportunistic improvements to be detected and suggested. The user would probably learn the advices quicker by reading a dedicated book.

Also, though the authors of EAVE recognise a knowledge gap between the programmer and the compiler, the primary focus is on fixing the source code rather than targeting the very user's knowledge. By informing exclusively on code hacks, the user is tied to a specific compiler/architecture. Since this platform is to evolve over the course of time, this solution is not viable in the long term.

Besides, the system limits analysis on a single loop, and only treats vectorisation. If the programmer were to run the analysis on the entire source code, with advices covering a broader spectrum, there would be thousands of messages output. It would probably be too many for the user to focus on, thus the need to introduce a filter.

c) Matlab Code Analyzer

This solution from MathWorks was suggested during one of the interviews, and is in my view the closest to the purpose of this Degree Project⁴. The messages focus on relevant problems, highlight the responsible code line and clearly state how to resolve it, while explaining the causes behind it (though not shown in Figure 4, some notifications display a button to expand the details).

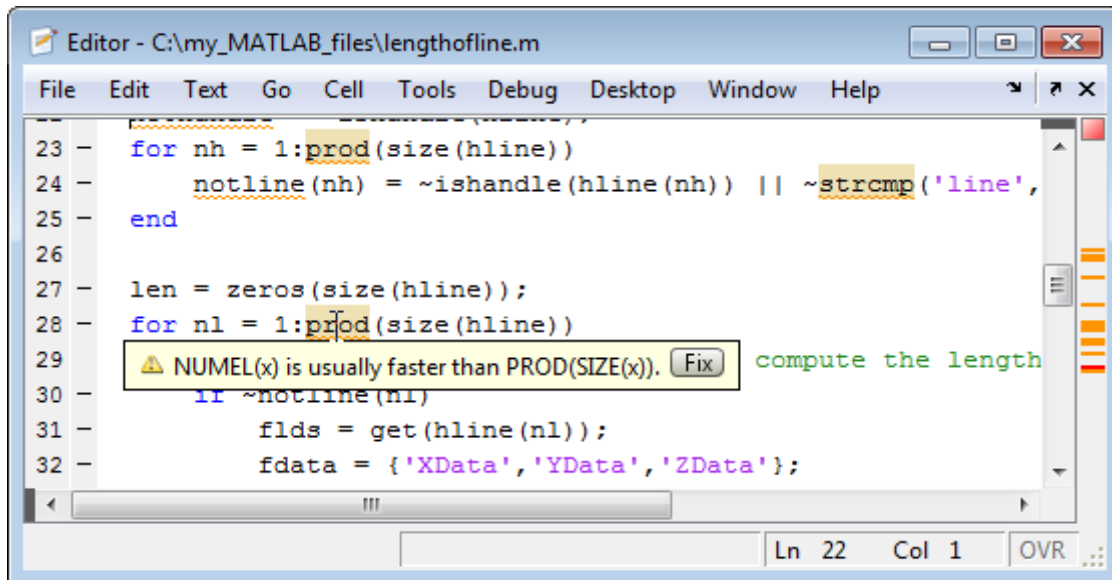


Figure 4: Example of message from Code Analyzer, as show on Matlab's website⁴

However, as with EAVE, there is no handling of the amount of messages. As a consequence, the assistant includes only the most important ones. Furthermore, it deprives itself from a simple feedback which could inform the user that a portion of code was fully understood.

In addition, the interface cannot query the programmer, nor can it be queried itself. Even a rudimentary polling of the user would enable setting parameters which cannot be expressed as code, as discussed in the Problem definition.

2.3. Rationale and differentiators

Thanks to the analysis of these similar works, a few differentiators were identified, so as to make this thesis a conceivable choice for real-world development environments.

a) The compiler queries the programmer

Interfaces such as VISTA and EAVE rely on their design to guide the programmers into optimising each and every aspect of their code, but without a filter to select the

⁴ http://www.mathworks.se/help/techdoc/matlab_env/brqxeeu-151.html#brqxeeu-155 (accessed 07.09.2012).

important transformations the interaction is lengthy. Matlab Code Analyzer does a little better by spontaneously enlightening the areas of interest, but focuses only on fixing issues in the code. This thesis proposes to go two steps further, by adding *positive* feedback and the possibility to query the programmer for precisions on a portion of code.

More specifically, interaction with the programmer can be divided into four distinct tasks. The compiler should be able to:

- *inform*: tells how well a portion of code was compiled, introduces a compilation technique, relates a coding practice, promotes a feature from the standard, etc.
- *alert*: incites the user to correct a supposed flaw, notifies about a potential vulnerability which could arise with further lack of attention, recommends a performance tweak. This is the task at hand in Code Analyzer. As opposed to the previous task, here we demand some code to be fixed. Also, the difference with standard unfiltered compiler warnings is that they concern code which might not execute with the intended meaning, though here only tuning of a working program is concerned.
- *ask*: inquires a clarification about a portion of code. Sometimes a warning is insufficient, when the clarification cannot be expressed by updating the code. In cases like choosing the implementation of a tree structure (*set* or *map* in C++) or the character set (Latin-9, UTF-8, etc.) of a *string* object, the interface could trigger radio buttons to query the programmer.
- *answer*: responds to a direct interrogation from the user. The requirement for artificial intelligence is not discussed in this document, though an attempt to test its design was made further in the second prototype.

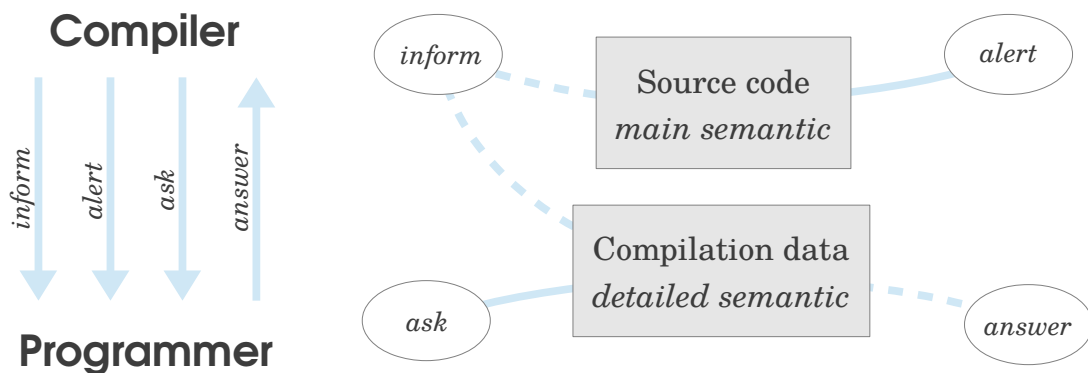


Figure 5: On the left image, arrows indicate who initiates the communication. On the right one, lines bind each task to which data is discussed (a dotted line indicates the data is not intended to be modified).

The third task is quite interesting, as it can relieve the user from the complexity of

tuning the whole program. Through a selection heuristic, the compiler could request intervention on the most critical parts of the program, setting the rest to default parameters. Furthermore, it would allow languages to be designed with two levels of semantic: main and detailed (Figure 5). The main semantic would be expressed as the source code itself, as with standard languages. The detailed semantic could be accessible through menus in the IDE, or comments in the code, and would be used by the selection heuristic to generate queries.

b) No separate interface

VISTA, EAVE, and to a greater extent profiling tools in general provide the performance analysis and suggestions of improvement as an interface distinct from the editor. This requires the users to learn how to use it, and this knowledge acts as a disincentive on their commitment to start profiling. Here we will make the messages appear besides the relevant source line, as in Code Analyzer.

No interface has to be learned here, the only limit to the user's will to tune the program is the clarity of the messages, which is discussed further in this thesis. Moreover, such an interface can be easily implemented. In practice a compiler like GCC already outputs the line number along with every warning and error, and an IDE such as Eclipse is able to display warnings and errors in relation to the targeted line. This also helps the context to be accurately identified.

c) Cooperation instead of assistance

VISTA, EAVE and to a lesser extent Code Analyzer illustrate what an *assisting* agent is: it waits for the user to request help, it does not query him/her, and it focuses on helping the user fix an issue rather than improving his/her knowledge. By contrast, a *cooperation* is similar to a discussion, in which both interlocutors can engage the conversation, ask questions and answer them. Moreover, there must be no assumption that the programmer is familiar with any of the concepts involved. Contrary to VISTA and EAVE which refer to the optimisation techniques by their names, here the tasks will give a short explanation and always cite their source, so that the user is never responsible for owning the proper reference.

This is actually not meant as a human-machine cooperation, since no artificial intelligence is intended. Instead, it is the designer behind the interaction tasks who is cooperating with the user. As observed in the Challenges, the feedback messages and the selection heuristic will embody the designer's point of view on how to improve a program. *Users Need Rationales*, as Carroll and Aaronson (1988) state, and a liberty for arguing is a decent mean to satisfy it.

d) The filtered notifications

In a simple technical design such as the first prototype shown further, or in Code Analyzer, a single pass on the code generates notifications to be displayed in the development environment. For simplicity, let us call messages the questions from the third task *ask* too. In EAVE and Code Analyzer, the suggestions generated all have critical importance, however we want to consider every possible feedback here. To avoid burdening the programmer with countless notifications, only a handful should be selected to be displayed at each build. Furthermore, provided the system remembers the messages already displayed so as to prefer new ones, the notifications could seem *spontaneous*. The programmer would then discover the suggestions at his/her own pace, skipping them when not ready.

Besides, for the sake of transparency and to allow users to retrieve missed feedback, the full list of published messages must be available, that is all the notifications generated, before they were filtered to keep a handful. This list, as well as the heuristic used to choose the displayable messages, are briefly covered in the Findings and suggestions for future work.

3. Preparatory study

At this point, a series of interviews was necessary in order to evaluate the users' preferences regarding the contents of the messages, and ensure there would be no clear rejection of “improvements” to compilers.

3.1. The interview plan

Since the questions were to refer to how people program, I chose to start with a simple algorithmic task. Being in familiar working conditions, the interviewees could then share their interrogations through a *think aloud*, and I would later ask how they usually manage their other programming projects.

The purposes of the interviews were:

- to poll the receptiveness of users regarding the introduction of a feedback, and their presumptions about how a discussion with the compiler could look like,
- to confirm that all users – even experts – have an incomplete knowledge about programming, which lets them miss a few opportunities of improvements,
- to gather the resources they rely on when they need to learn,
- to observe their use of their own development environment and reflect on how to integrate a compiler feedback,
- to query the feasibility of an embodied agent to support the communication.

For this study I needed participants with experience in programming. I thus selected KTH peers whom I knew had such experience. Each interview would be conducted on a platform the participant would be familiar with, be it his/her laptop or a school desktop computer. I would sit next to the interviewee and give the instructions and questions while he/she had the IDE in sight. This proximity was meant for an open discussion to help the think aloud. To compensate the possible stress of having someone watch over their shoulder, the problems were overly simple, and I would insist on the absence of competition or comparison of performance between the different interviews.

Three problems were written, so as to cover a broader range of expertise, on three *distinct* typical goals in programming: *performance*, *security* and *maintainability/extendibility*. The interviewees were to solve the problem first without knowledge of the goal, then they were asked how they would optimise it according to the corresponding goal. The problems would be given in any order, usually two in an interview, so as to fit in 40 minutes. The performance problem was always given, and

the second was chosen to match the experience of the interviewee so that he/she could solve it quickly. Refer to Appendix B for the interview sheet actually used.

For the performance version, the code was initially intended to contain: a loop or nested loops (to expose the many possible optimisations related to loops), a multiplication by the loop index or a constant (to expose Strength Reduction techniques), and the possibility to reuse some intermediary computations.

The problem statement became: *Write a function which receives a 100x100 array t and three integers a, b and c, and compute $t[i][j] = \frac{i \cdot 2^a + j \cdot b}{c}$* (See Figure 6).

```
void init(int t[100][100], int a, int b, int c) {  
    for (int i = 0; i < 100; i++) {  
        for (int j = 0; j < 100; j++) {  
            t[i][j] = (i * pow(2, a) + j * b) / c;  
        }  
    }  
}
```

Figure 6: Example solution written in C

For the security version, the code was initially intended to contain: a buffer copy with assertion on the size (to expose a possibility for Buffer Overflow), the call of an untrusted function (to expose the checks for inputs and outputs), the need for a random number (to expose the weaknesses of pseudorandom number generators), and an integer computation which could overflow.

The problem was finally stated as: *You are writing a simple login program. Start by reading a 64-character ID and a checksum from standard input. Then call `login(ID)`, which returns the user name found in database. At last, compute the sum of its characters and compare it to the checksum. If it matches, print “Welcome [user]”* (See figure 7).

```
char ID[65];  
int checksum;  
scanf("%s %d", ID, &checksum);  
char* user = login(ID);  
int sum = 0;  
for (char* c = user; *c != 0; c++)  
    sum += *c;  
if (sum == checksum)  
    printf("Welcome %s\n", user);
```

Figure 7: Example solution written in C

For the maintainability/extensibility version, the interviewee would design a class, which was initially intended to contain: the need for 64-bit variables, the storage of a string (to expose the choice to make it statically or dynamically sized), the need to store booleans (either in separate variables, or in a dedicated object), the use of constants (which can at least be defined either as compile-time or run-time constants), the implementation of optional fields (to expose the use of common fields, or class polymorphism), and the possibility to use different error models (either by functions returning error codes, or the use of exceptions).

The statement for the problem became: *Your client is a web site locating all shopping places in the world. Your task is to design the data structure(s) needed to store them. Only a subset of the typical fields is required: a country code, the name, whether it has a shoe shop / restaurant / barber shop, and whether it is a mall or a shopping street. For malls, we store the number of shoppers per year, and for shopping streets the delimiting street numbers* (See Figure 8).

```
class Shopping_place {
    enum { GBR=44, SWE=46 } country_code;
    char name[100];
    bitset<3> contains;
};
class Mall: public Shopping_place {
    long long shoppers_per_year;
};
class Shopping_street: public Shopping_place {
    int street_begin, street_end;
};
```

Figure 8: Example solution written in C++

3.2. Conducting the interviews

Five interviews were carried over a month at KTH, each one lasting from 40 minutes to an hour. The interviewees had pretty different profiles, as shown in Table 1. This broad range helped to mitigate the low number of interviews which was due to the difficulty to schedule such a lengthy meeting.

The first finding to note is the average programming experience of 9 years, and 2,5 years on average in their current language. None of the participants were amateurs, and this gives weight to the persistence of misconceptions and interrogations shown further.

	Aurélien	Alexandre	Mikael	Léo	Siim
Field	Numerical Analysis	Networking	Software Engineering	Cryptography	Robotics
Language (IDE)	C++ (Eclipse)	Java (Eclipse/Geany)	C++ (Vim & GCC)	C/C++ (emacs & GCC)	Delphi (CodeGear)
Experience (in main language)	8 years (4 years)	10 years (6 months)	11~12 years (9~10 years)	4 years (3 years)	11 years (3 years)
Target when programming	readability, simple design, durability in the long term	working code, readability, simple code for maintainability	re-usability and extensibility	readability, maintenance, factorising	working code, readability, extensibility, simple design
Can improve his projects? (himself?)	yes (yes)	yes, with more time (yes)	yes (yes)	yes, also spend more time (yes)	definitely, and time also (yes)
Resources browsed	C++ standard, books, Google, Sun guidelines, Parashift FAQ, articles	courses, books, communities of good practices like Symphony	wandering on the Internet for small fixes	Google, Stack Overflow	a good book on Delphi, hands-on, Google
Preferred interaction	compiler's output read by the IDE	underlining and tooltips in the code	he initiates it, tree-like conversation	enabled with a flag, suggests after compiles	messages like in Matlab, with a Fix button

Table 1: Summary of the results from the interviews

a) Problem solving

The Performance problem was given to five participants, the Security problem to three participants and the Extensibility problem to two participants. Since I was present to give further explanations, the instructions were well understood, the interviewees writing a quick first draft before trying to optimise it.

When asked to optimise the first problem, only one participant proposed a systematic approach, namely taking advantage of the variables' properties (mostly constness here), and analysing the assembly output. All the others resorted to guessing tips and tricks. Another participant was aware of cache locality and the influence of looping order, but could not tell how to improve it. Surprisingly, only one interviewee knew the binary shift trick for multiplication by a power of two (replacing `i*pow(2, a)` by `i<<a`). Some had difficulties to remember the proper syntax and library calls, as for the `C pow` function, or how to pass an array as a function argument.

Though not presented as such, the second problem was meant as a collection of traps for the participant to find. The success was *not* to be measured by how many were found though, as a single weakness compromises the whole program. Only the student in Security found them all with a generic approach, namely securing the

communication with the database, using a cryptographic checksum, checking the inputs and multiplying the layers of security.

The third problem was a little harder to exploit. Lengthier to explain in practice, it was given to too few participants. Consequently I could only gather a few interrogations to be answered among the messages.

One could argue that asking a quick working draft then its optimisation induced the production of sub-efficient code in all three problems. This is however meant as a reflection of the IT industry. Indeed, delivery of working software under tight schedule is at the core of the Agile development method⁵, for example. The interviews showed that people perform hardly well at optimising code, the interface should thus provide help to produce *efficient working code* at first draft.

It also appeared that proper optimisation is not barely a matter of time. One *has to be* an expert in the specific field corresponding to the specific aspect targeted. This leads to projects centred around one aspect, the others becoming sub-efficient. A perfect example is the BSD family of operating systems: FreeBSD (performance), OpenBSD (security), NetBSD (portability)⁶. The interface should thus help to compensate where users lack expertise.

b) Subsequent findings

To the question about the usual target when programming, readability was the most common answer. “Code that just works” was favoured by two interviewees. Apart from these, there were very different goals, as expected.

At many times I asked the interviewees how they thought the compiler was carrying an operation, and they showed interest in the answer as they had already been wondering it. Such questions were for example: “Does the compiler automatically unroll loops? At which optimisation level?”, “Does it enable protection against buffer overflow?”, “Does it actually store constants as memory variables?”. The existence of such unanswered interrogations which somehow *haunt* the users, calls for serendipitous information retrieval (De Bruijn & Spence, 2008). The interface should output several different messages at each execution and cite a source in each one, so as to expose the user to much information, that potentially answers a dormant interrogation. Moreover, for the same purpose the messages should be the shortest, and the number of sources limited to one.

All participants admitted that they could still improve their projects, and that they

5 Refer to the first, third and seventh principles at <http://www.agilealliance.org/the-alliance/the-agile-manifesto/the-twelve-principles-of-agile-software/> (accessed 07.09.2012).

6 For a brief history and comparison of the three systems, see <http://www.freebsdworld.gr/freebsd/bsd-family-tree.html> (accessed 07.09.2012).

had to improve themselves. Many use programming books as references, and all use Internet for occasional problems. The goal of this question was to observe how much the Web is used as a programming reference. While there is no control over the quality of the advices found there, the interface could use the source links in the notifications to reference serious documents.

The interviewees were neutral regarding the possibility to be queried by the compiler, but mostly opposed the idea to embody it, since it would not look like a serious interface.

When observing those using an IDE, I noticed they were all disabling the *Message of the Day* tooltip at startup. The reasons they gave were:

- Much of the information displayed documents basic functionalities.
- The first few messages are not teaching anything.
- They are not contextual, nor are they relevant.

The first note motivated the requirement for technicality of the messages, that is they should always seem relevant, not worth being disabled, even if they will be quite complex. The source link can then provide the necessary details to users willing to follow it. As for the third note, it instructs to avoid citing what the compiler *can do* in general, in favour of informing what it *will do* on a particular line of code. Thereby, the feedback is closest to the context which triggered it.

4. Three prototypes

As advocated in Dow et al. (2010), I chose to design several prototypes in parallel, each embodying a distinct approach to the solution. This work does not include their iterations though, which are discussed in the next chapter.

4.1. First prototype: a stripped version for GCC

As part of this project, I seized the opportunity to imagine how would a real-world compiler be modified, by applying for a Google Summer of Code for GCC. The purpose was to confront the actual difficulties which can arise when attempting to implement such an interface. Below is the proposition submitted to the mentors at GCC:

Title: Provide optimizations feedback through post-compilation messages

GCC currently provides no concise way to inform the user whether it applied an expected optimization (i.e. it "understood" the code). As a result, some will do premature optimizations when they do not trust the compiler, and some others will create overly convoluted code with blind belief in the compiler. This is especially relevant for users non-initiated to the internals of GCC.

The project I would like to propose is a feedback for the optimizations performed by GCC. To avoid binding users to the compiler, I would focus on some very standard optimizations across vendors, or for some specific yet nice features I would indicate their specificity to GCC/an architecture.

The feedback would be triggered when compilation is successful, and display a couple of different messages each time it is run:

```
gcc --feedback test.c
test.c:xx:x: info: All operands being constant, constant folding was applied to assign
'2560' to 'a'
test.c:xx:x: info: GCC could not fold constants here because...
test.c:xx:x: info: As integers are stored in binary format, strength reduction was
applied to replace '* 8' by '<< 3'
test.c:xx:x: info: Basic block vectorization was applied to pack the 3 independent
additions into a single SIMD instruction
test.c:xx:x: info: GCC implements unordered_map as open-addressed hash tables, with
double hashing probing
```

As a difference with the internal verbose messages, here they would form a set, and the system would remember those already displayed and decrease their frequency of occurrence between compilations. All messages would explain what triggered them, cite the optimization name, and describe the consequence.

Though optimizations are the most obvious purpose of the feedback messages, the project has a broader scope: output any relevant short piece of information (may it be the implementation of STL containers, or putting light on an unknown aspect of the standard for example).

As for the work plan, it would consist in:

- _ Enumerating all possible messages in the messages set.
- _ Implementing a function receiving feedback from each optimization unit and choosing whether to display it:
`info_printf(enum INFO_INDEX, const char*, ...);`
- _ Write a formatting guide for adding messages in the set.

Figure 9: My Google Summer of Code proposition which led to the first prototype

Without a proof of concept in terms of code, the proposition was rejected. Nevertheless, the challenges having been identified and a working solution proposed, a prototype was finalised, using HTML5, CSS3 and JavaScript for the neat look and interactivity they provide⁷ (Figures 10 and 11). It represents the IDE frame where code is edited, and focusses on the interaction tasks *inform* and *alert*, both identified by two distinct margin icons. Note that the sample messages presented in all three prototypes are not meant to be true for any particular compiler, they simply look technical and precise.

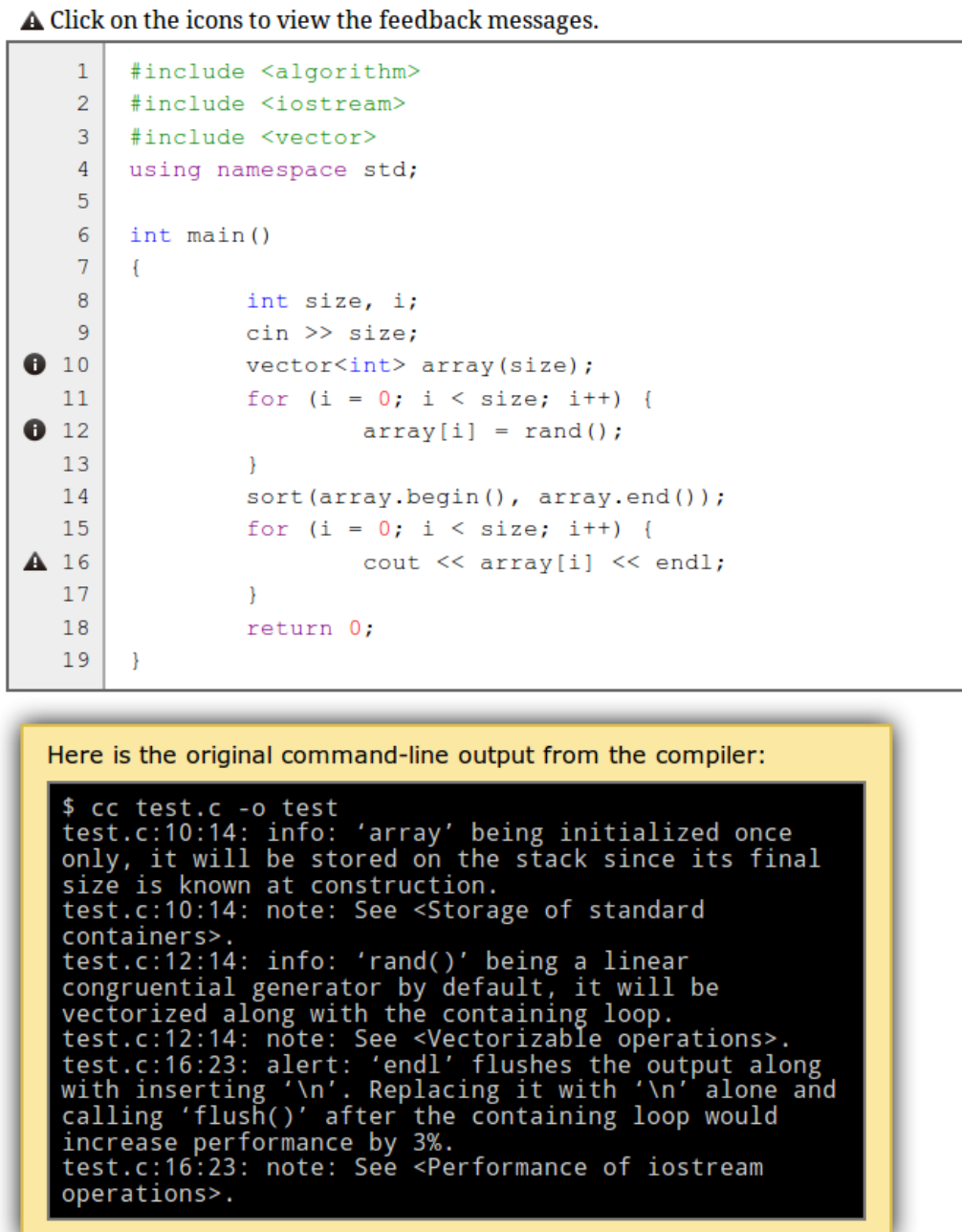


Figure 10: Screen from the first prototype⁷ showing the messages in command-line output before they are rendered by the IDE.

⁷ The first prototype is available online at <http://www.csc.kth.se/~traf/thesis/proto1.html>.

▲ Click on the icons to view the feedback messages.

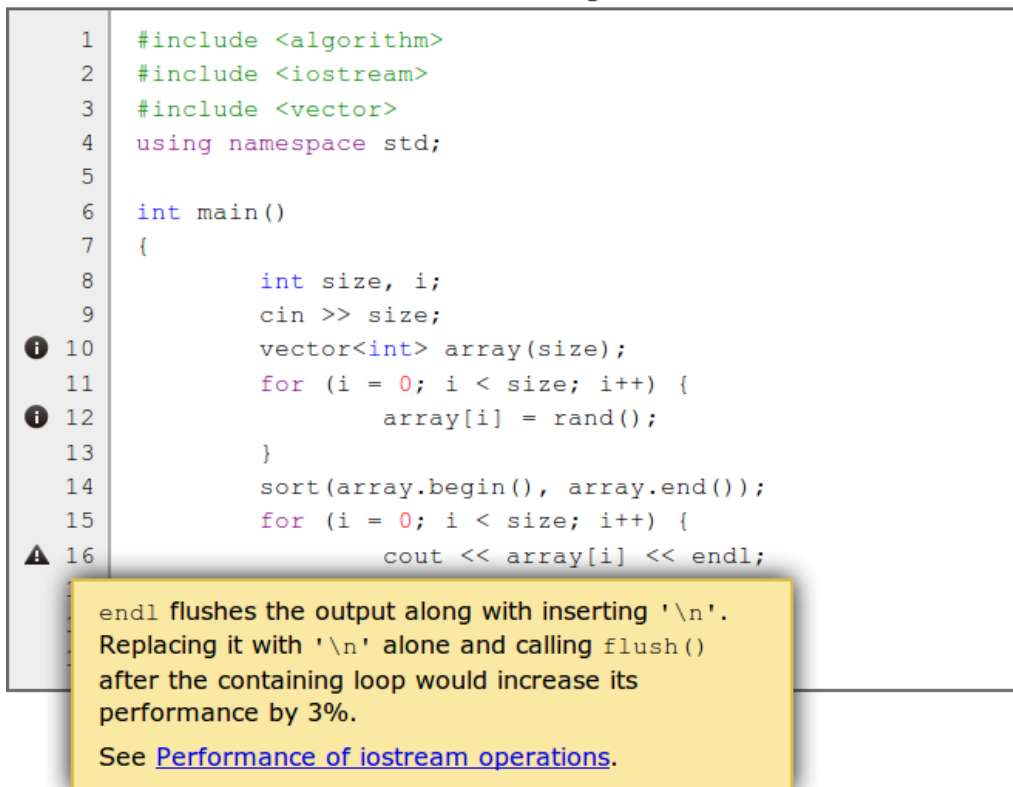


Figure 11: Screen from the first prototype showing an alert message.

Relating each notification to the relevant source code line is not usually a problem. All major executable file formats being able to store debugging information including line numbers⁸, a compiler such as GCC keeps track of the original line numbers at any time.

There are two approaches to make the compiler generate the messages. The first one is suggested in the proposal in Figure 9. It would consist in having each optimisation unit store its own messages and output them when it executes, using a dedicated `info_printf` function. This has the advantage to provide the best precision of feedback: the compiler informs on its operation at the *very moment* it does it, without speculation. However, the messages are then tangled inside the compiler's code, preventing the addition of new ones and impeding any desirable transparency of operation.

The second approach would consist in storing all possible messages separately from the compiler. Each notification would then be stored as a pair `{text, trigger}`, the latter being a condition on each instruction processed which enables the output of the

⁸ For PE/COFF (.exe) format, refer to The .debug Section in <http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx>, for ELF (UNIX), see Line Number Information in the DWARF specification at <http://dwarfstd.org/doc/DWARF4.pdf>, and for its ancestor Stabs at <http://sourceware.org/gdb/current/onlinedocs/stabs.html#Line-Numbers> (all accessed 04.09.2012).

corresponding *text*. It would require the definition of a syntax for *trigger*, which is beyond the scope of this thesis. This approach is the one I would recommend though, as it would allow the messages to be written for several compilers, and storing them in files would permit their sharing among users, as discussed further.

The most important task along with designing the interface was to provide an exhaustive list of possible messages, to give a clearer idea of its usefulness. In order to manage the enumeration, I focused on the standard optimisations performed among most modern software:

- Register Allocation (explaining how faster an operation is performed in registers, telling whether for an inner loop or a function all automatic variables could be stored in registers or spilling happened, citing the allocation technique used, informing which conditions allow a data structure to be stored in registers)
- Strength Reduction (indicating when a multiplication by a loop index was carried with an addition, warning about the use of floating point functions on integers and propose alternatives, showing the replacement of multiplications with powers of 2 by binary shifts)
- Common Sub-expression Elimination (informing where an expression was found to be redundant and how the code was replaced, enumerating which operations are taken into account in CSE)
- Value Range Propagation (telling when a constant has been properly propagated, showing that the detected range of values for a variable leads to a performance gain, enumerating which types can be propagated by the compiler, citing the Static Single Assignment technique)
- Branch Prediction (informing when a dead section was detected and will not be compiled, showing how branch probabilities translate into code and how performance improves, suggesting the use of a profiler)
- Functions Optimisations (telling when and why a particular function was inlined, informing about the compilation flags toggling inlining, warning when too many variables are passed to a function as the registers are limited, telling whether Tail Recursion could be applied on a recursive function, describing how complex objects like classes are passed as arguments and returned)
- Data Alignment (explaining why the size of a structure can be bigger than the sum of its fields' sizes, telling in which case padding was added inside a data structure, informing about the performance penalty when accessing unaligned data)

- Stack Layout (pointing which variables are stored on the stack, giving figures as to how performance increases with such storage, proposing buffer overflow protection techniques and informing which flags enable them, giving the typical stack size on the target system)
- Vectorisation / Parallelisation (indicating whether and why a loop could be vectorised on a SIMD-capable architecture, describing how to best control vectorisation through flags and tools, providing figures as to how performance increased on a loop with the use of SIMD instructions, telling whether several similar operations could be packed in a single instruction, suggesting parallelisation libraries to execute simultaneous iterations on parallel threads)

I also focused on the various aspects of a language design (mainly C++) to enumerate a few more topics for feedback messages:

- Manipulation of files (explaining the difference in performance/cache use/security between the various input/output functions, pointing out the origins of buffer overflows and the means to avoid them)
- Time management (warning about the year 2038 problem and discussing means to circumvent it)
- Strings and characters (providing a comparison between null-terminated and sized strings, explaining how the fast character testing and copying functions translate into code)
- The use of assertions (giving figures as to how enabling assertions impact performance, telling whether assertions are used for value range analysis)
- Style and formatting (warning when a function is too big that it would not fit in a cache, citing which character set was detected for the source file and how strings are stored in the output, here the messages can greatly depend on the designer)
- Run-time checks (enumerating the list of available checks and the flags enabling them, informing when such checks have been inserted and their cost)
- Classes (showing when constructors and destructors are inlined, giving the number of system calls involved in the use of dynamically sized objects, describing the actual implementation of standard complex classes like bit-fields or hash tables, telling which operations will leave a certain iterator stable)
- Functions (introducing the overhead of a function call, describing which registers a particular function saves/uses)
- Data storage (explaining where in memory a particular static/automatic

variable will be stored, informing where in memory constants are saved, introducing endianness and why one should care about it, suggesting faster initialisation methods like `memset`)

- Floating point types (warning about the use of equality with such variables, describing the range of acceptable values including subnormal numbers and the expectable precisions)
- Operators (telling how certain ambiguous operations like integer division behave with negative operands, comparing the speed of an addition versus a multiplication on the target architecture, warning when an apparently small operation like a norm has a non-negligible cost, informing about the possibility of integer overflow and proposing various means to avoid it)
- Control flow structures (explaining how switch statements are converted into code and their performance benefit)
- Exception handling (introducing how this mechanism is translated into code, citing which types of exceptions are the most easily dealt with by the compiler)

These lists are certainly not exhaustive but already give a strong basis of feedback messages the interface could implement.

4.2. Second prototype: a communicative compiler

A second prototype was designed to complement the feedback side with the possibility to query and discuss with the programmer, corresponding to the interaction tasks *ask* and *answer*. It requires a dedicated frame in the IDE's workspace to display the queries⁹ (Figures 12, 13 and 14).

i The second frame is dedicated to the discussion with the compiler.

The screenshot shows a C++ code editor with the following code:

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  int main()
7  {
8      int size, i;
9      cin >> size;
10     vector<int> array(size);
11     for (i = 0; i < size; i++) {
12         array[i] = rand();
13     }
14     sort(array.begin(), array.end());
15     for (i = 0; i < size; i++) {
16         cout << array[i] << endl;
17     }
18     return 0;
19 }
```

Below the code editor is a query dialog box with the following text:

To improve performance, the algorithm reading integers can be tuned to match the properties of the integer string (See [Input routines](#)):

☐ binary ☒ positive or zero

☐ octal

☒ decimal

☐ decimal/prefixed

☐ hexadecimal

A blue button labeled "PROCEED" is located to the right of the radio buttons.

Figure 12: Screen from the second prototype⁹, showing a query spontaneously submitted by the compiler.

After a successful compilation, the second frame displays a set of spontaneously generated queries. As with the first prototype, these notifications will change each time a compilation is run. Answering them is never required, they will default to safe values.

Technically, this prototype requires a compiler-specific semantic to express the answers to queries. As an example, clicking “Proceed” on the question in Figure 13

⁹ The second prototype is available online at <http://www.csc.kth.se/~traf/thesis/proto2.html>.

could add `#pragma rand subtract_with_carry` before line 12, effectively giving this hint for the next build run.

Triggering these messages would then be similar to the first prototype. With the second approach in mind, they would form the triplets $\{text, trigger, pragma\}$, where *text* would be formatted to generate a query, and *pragma* would contain the text added to the source code after answering the question.

Furthermore, the messages could be independent from the compiler if such *detailed* semantic was part of the language standard, as advocated in the previous section Rationale and differentiators. They would then benefit from the same possibility of sharing among users as in the first prototype.

i The second frame is dedicated to the discussion with the compiler.

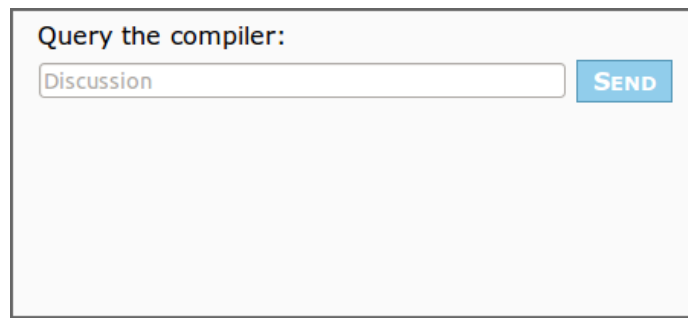
```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  int main()
7  {
8      int size, i;
9      cin >> size;
10     vector<int> array(size);
11     for (i = 0; i < size; i++) {
12         array[i] = rand();
13     }
14     sort(array.begin(), array.end());
15     for (i = 0; i < size; i++) {
16         cout << array[i] << endl;
17     }
18     return 0;
19 }
```

To improve security, the implementation of `rand()` can be chosen among (See [Pseudorandom number generators](#)):

- ☐ linear congruential generator (vectorizable)
- ☒ subtract-with-carry generator (fast)
- ☐ Mersenne twister generator (medium)

PROCEED

Figure 13: It is also possible to click the elements in the editor to edit them, and bypass the suggestions if unwanted.



Query the compiler:

Discussion

SEND

Figure 14: The last suggested frame shows a field for discussion with the compiler. The talkbot behind it is not configured to be functional though.

Enumerating the messages to include in such an interface is not as straightforward as for the first prototype. It requires seeking the aspects of a language semantics which are incomplete. For C++, I focused on the aspects which would benefit from the increased expressiveness without burdening the main semantic:

- Choosing the character encoding of a string or stream, which will influence functions such as `strlen` or `isspace`
- Setting the locale of the program, since in the current C standard a single library is dedicated to it
- Asserting that a certain variable will never overflow, which could enable certain optimisations
- Choosing the algorithm behind certain mathematical operations such as computing the inverse square root, while giving the precision of each
- Querying the expectable branching probabilities in a critical portion of code
- Asking whether to maintain an assertion in release mode when some sample cases show its failure
- Gathering the properties of an variable read from a stream, to enable the use of faster routines
- Selecting the algorithm to sort an array, the default being usually Quick Sort
- Setting the precision for the storage of time or delays
- Selecting the implementation method to generate random numbers (in C++ a library is dedicated to it, though in C it is a single function)
- Choosing the underlying storage of an array of bits (in C++ it is stored as a `bitset` though it sacrifices performance in comparison with an array of integers)
- Setting the properties of a container (the implemented directions of iteration, whether the size often increases, where items are appended, whether stable

iterators are required)

- Asking for the implementation method of a binary tree or a hash table
- Proposing the stack protection method against memory corruption including buffer overflows

These semantics being optional, they must not have critical importance on the program. They will rather be set to tune its various aspects, when the program was already proven to work properly. As with the previous prototype, this list is certainly not exhaustive, but it gives an insight for the usefulness of the *querying* improvement.

4.3. Third prototype: a far-fetched alternative

This prototype goes one step further in the coupling between the compiler and its interface¹⁰ (Figures 15 and 16). It uses a second frame to graphically represent the compiler's understanding of the various elements found in the code. It was mostly intended as a place for open suggestions from the testers, and is not to be matched with the interaction tasks previously mentioned.

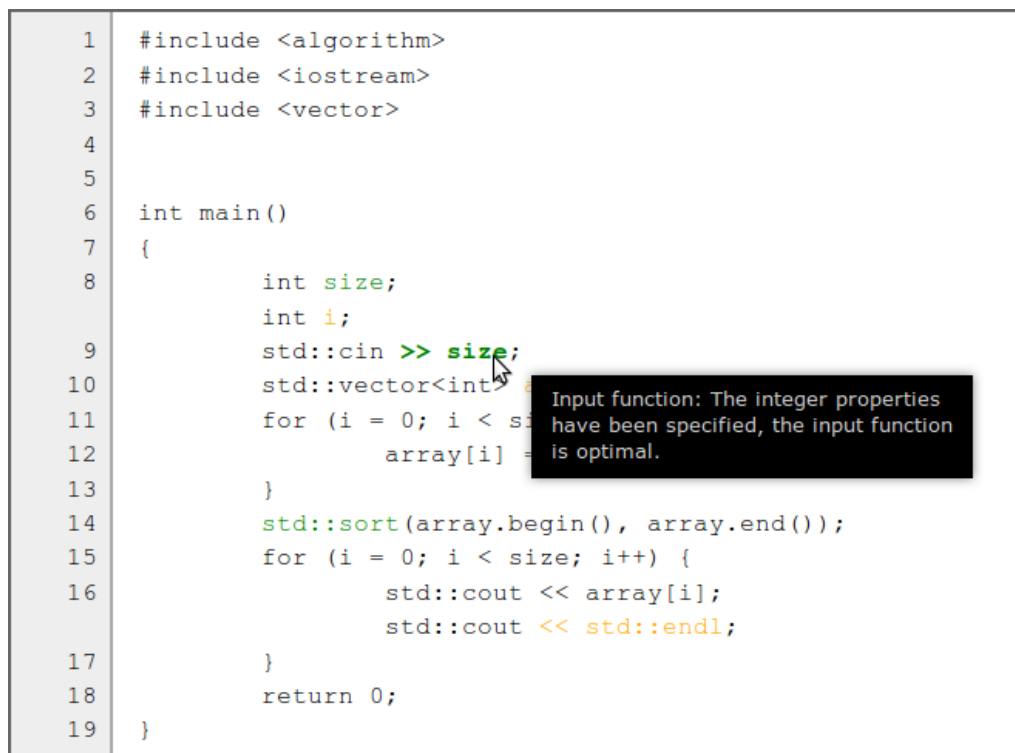


Figure 15: Screen from the third prototype's second frame¹⁰.

Having noticed in the interviews that users had a will to do good despite their refusal of an embodied interface, I chose to depict the compiler as a living system. The

¹⁰ The third prototype is available online at <http://www.csc.kth.se/~traf/thesis/proto3.html>.

interaction then consists in the user helping the compiler understand the code, a simple colour scheme being used to inform how an element is apprehended. The two frames are representing the code textually, however the bottom one *should* evolve towards a more suitable representation, such as a coloured dataflow diagram.

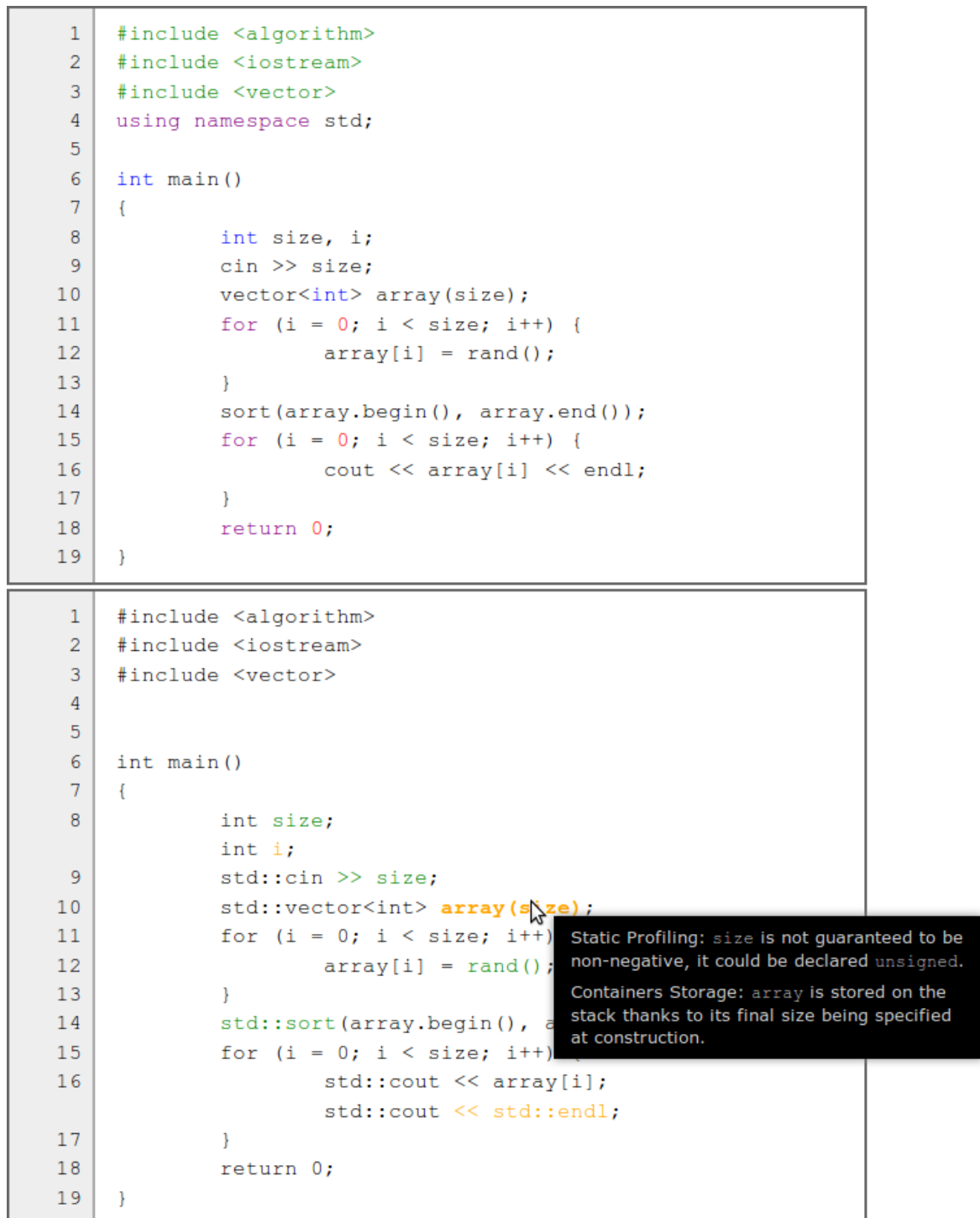


Figure 16: The colours were banned from tooltips, so as not to become too invasive.

5. User study with the three prototypes

After creating the parallel prototypes, a quick round of testing would give further direction as to how they should evolve, as well as a few technical precisions.

5.1. A new round of interviews

This new series of interviews was intended to ensure the goals set after the first gathering had been met, and estimate how users were considering the value added. In order to gather more participants, the testing would last 20 minutes. The requirement was no longer that participants had a decent experience in programming, they would instead have familiarity with an IDE, and understand basic C++. In addition, to avoid considering the influence of being already familiar with compiler feedback, I chose to interview participants different from the preparatory study. The context was identical however, each interview taking place at KTH, while I would sit next to the participant and ask questions about the prototypes at sight. After testing the three prototypes in order, the interviewee would choose his/her favourite and explain why, then answer a few last questions about the use of *personae* (see the next dedicated section). Refer to Figure 17 and Appendix B for the test sheet used. Choosing to test the prototypes *in order* was actually meant to help introducing the idea of compiler feedback. Indeed, it was a concept rather than a finished prototype which was tested here.



Figure 17: The panoply of a tester hunter

Five tests were conducted over two weeks, polling random students at KTH. Both the first and the third prototype were praised, the former for the technical insights, the latter as a quick overview of the compiler's job. As with the preparatory study, few of them were initially showing interest in a feedback from the compiler. I had to pursue the description to the personae, until the concept of compiler feedback became clear and coherent. Then they all agreed that they would not disable it like a *Message of the Day*, which was my main concern. In addition, some expressed they were actually looking forward to seeing a working release in the future.

On the downside, the second prototype was generally little understood. This might have been influenced by the unusual situation of being queried by the compiler. However, in my opinion it was the presentation as a separate frame which made it difficult to spot the context at hand, that is which part of the program the question was dealing with. The last discussion frame, corresponding to the interaction task *answer*, received the same reception. For lack of example questions and because of the unrelated answers the talkbot was returning, this aspect of the prototype was not understood. The integration of queries as in the first prototype then remains to be tested for future iterations.

5.2. Findings and suggestions for future work

Thanks to the feedback gathered from the testers, a few additions are proposed for future work on the prototypes.

a) The personae

This idea appeared with the possibility to store messages in files, as discussed in the first and second prototypes. Provided a *trigger* syntax is defined, each notification can be stored aside the compiler, under a triplet *{text, trigger, [pragma]}*. Sets of notifications can then be stored as files, forming categories of similarly related items. The addition of a field along every message of the first prototype could further allow the programmer to be aware of the category at hand, and increase or decrease its further occurrences, in order to receive the most interesting feedbacks.

Categories form the default set of messages shipped with the compiler. To extend and customise this set, users could create their own files and share them. The idea behind *personae* here is to bind an author to a file with notifications. Knowing who wrote a certain suggestion could give value to it and mitigate the effect of a poor feedback, particularly if the author is known for being a good programmer. Here, a simple and recommended way to store the category and author's names is through the file's name.

In practice most testers were very receptive to it, with different intents. One tester did not care about the author's name, as long as he/she was a specialist. Another one conceived the sharing of files inside teams of developers, in companies. The last considered contributing in online communities of developers rather than friends.

b) A few rules for composing the messages

During the tests I emphasised the questioning on the quality of the messages, and how their redaction should be improved. While the testers were often puzzled with the feedback's technicality, they were paradoxically very fine with it. Indeed, two actually argued that they were used to this situation. The links to references were intended to balance this complexity, and in practice were praised by all interviewees. The quality of redaction had a great influence on the participants' reception of each message, though. The first query in the second prototype, for example, was systematically deemed too complex, and I always had to explain it. This difficulty motivated further the addition of personae, to let programmers choose a good teacher, and sketch a set of rules to help the redaction of further messages:

- *Technicality*: The feedback should rather be too technical than not enough, and provide a substantial benefit which will be highlighted.
- *Context*: Indicate what the compiler *will do* for a particular line/object rather than what it *can do* in general.
- *Referencing*: Cite one and only one source giving details for the corresponding feedback.
- *Neutrality*: Balance the amount of positive and negative feedback, that is when a line was well understood or when it needs tuning.
- *Clarity*: To be read and understood quickly, each message should receive careful attention and go straight to the point.

c) Transparency is crucial

As advocated in Sinha & Swearingen (2002), transparency has been a key design choice along this work. Providing a reference link along each feedback, targeting the programmer's knowledge rather than hacking tips, binding an author to the messages, defining a syntax for notifications and storing them in text files directly accessible to the programmer, were all choices motivated with transparency in mind. Furthermore, the next section presents a heuristic to filter messages, simple enough to be exposed to the user. I should be noted here that transparency is preferred over translucency – selecting what is shown and what is not – since no *thought* restriction on the

information given was intended. The latter could arise in the future though, with further iterations on the prototypes.

In my opinion, transparency is the key means to show and insist on the absence of artificial intelligence, or “smart assistant”, to govern the suggestions. Giving the qualifier *smart* to the robot could make users feel it is asserted to be smarter than them. Carroll and Aaronson (1988) bring out many receptiveness issues from interacting with such an agent, which transparency would greatly mitigate. Indeed, with access to the database of possible messages, and knowledge of who lies behind them, users are aware of the bounds of the compiler's intelligence, and will not expect more than it can actually help.

d) A proposed formula for the sorting and filtering of messages

To allow filtering the notifications while keeping a complete viewable list aside, an importance factor is to be computed for each message, to sort the list and output the first few. This factor can further be useful for the third prototype, to sort the paragraphs inside each tooltip.

For the purpose of being exposed to the programmer, the formula constructed here aims at simplicity. As a side effect, this would actually ease its implementation. The importance of a message should depend on the number of times it already occurred, on the number of messages already published on its line, on the user preference about the category of the notification, and on how critical it is for the line of code it refers to.

Let us note n the number of previous occurrences of the message, m the number of previous references to its line, f_c the factor assigned to its category, and f_l the local factor denoting its importance for the source code. A candidate formula recommended here would be the average of 2^{-n} , 2^{-m} , f_c and f_l , provided the two latter are bounded by $[0; 1]$. The strength of this expression is the simplicity to graphically represent an average. As a drawback, it does not allow completely disabling one category, though this is actually possible by simply deleting the corresponding file. Also, the factors averaged might have to receive an additional scale, which estimation is left to implementation.

Note that the reinitialisation of the heuristic is to be taken into account. The compiler might be reinstalled often, possibly clearing the memory of previously displayed messages and preferences. The more files/categories are stored, the more time will be needed to reach their previous importance values (considering the programmer can only decrease/increase this factor on every message received). The number of categories should thus be limited to a dozen on average.

6. Discussion and concluding words

As shown in the tests, the introduction of a feedback from the compiler was well received, either indirectly for the “big picture” overview it would provide, or for its relevant technical insights. Using this interface does require very little learning, if any, which is in my opinion a cornerstone of this work.

As for the direction to give to the prospective future iterations of the prototypes, since both the first and third prototypes were deemed promising the focus should be laid on implementing the common basis, namely generating the feedback messages. An option could then be available to choose *how* to display them. Furthermore, this choice might depend on the progress of the coding project. At the early stages when raw code is written, a few technical messages targeting the most critical aspects would be needed, as in the first prototype. Later in the process when these fragments are assembled, a broader overview like the third prototype would become useful.

6.1. Limitations

A few points were left aside during this work, either by lack of time or because they were a matter of debate. Among them, the initialisation of the system should be mentioned. Indeed, the introduction to the feedback is important, so that users do not disable it instinctively like a *Message of the day*. An example for an introductory text could be: *This compiler can output feedback messages telling how it understands the code, as well as technical suggestions. The list of feedback messages it can generate is contained in [folder], and can be extended by adding .cfb files downloaded from trusted authors.*

Also, this thesis work does not cover how to identify the level of knowledge of the users, as receiving too complex/simple feedback might eventually annoy them. While they can choose and download the notifications' files to add to the compiler, the initial set of categories could be specifically tailored to each one's knowledge, by estimating it with a question along the introductory text, for example.

One last issue which might eventually arise with the possibility to share authored feedback files is the lack of secure signing. If the author's name is stored in the name of the file as suggested in this document, nothing prevents it to be overwritten, or a wrong set of messages be imputed to the same author. The rationale behind the choice in this thesis is similar to the availability of coding guidelines online though: it is the user's responsibility to fetch the file at the source she/he trusts.

6.2. Personal conclusions

This project has been a challenging work for me. I started with the idea to have the compiler return performance-helper messages, much as in the first prototype. The expected design was then pretty clear, though the content of the messages remained to be defined. However, most of the time was actually spent on communication tasks. Indeed, my work suffered from the difficulty to clearly state what was intended by a *feedback*, because this word can be interpreted quite at will. From the initial sketch to the Google Summer of Code discussions, I invested many efforts in iteratively clarifying the purpose of this thesis. Designing the prototypes with the precision of HTML and CSS helped dramatically to show *exactly* what was intended.

The second challenge over this thesis was the difficulty to achieve a stable schedule. Due to the lack of an office to work in, I had to rely on my own motivation. The pace of work was thus extremely variable, though it never stopped. During the second half of the thesis, the regular meetings with my supervisor became really useful to impose milestones and keep the project progressing. Nevertheless, it lasted longer than expected, which in turn made the initial concept evolve widely. This is reflected in the structure of this document, starting from a simple feedback, to a discussion, and eventually to the personae and sharing of messages.

The last challenge was the management of the two rounds of interviews. In this thesis, they occupy a predominant place because a sheer amount of time was dedicated to them, especially for preparing the interview plans and the algorithmic tasks. My lack of experience to find participants made the first round last over a month, with relatively few interviews conducted. With the experience gained (and the help of home-baked cookies and muffins), the second round was more successful, though with paradoxically few participants since conducted during summer holidays.

In my view, it is the user studies which could be improved the most. I was satisfied with the preparation of the interview plans, particularly for the preparatory study, which probably allowed a pretty authentic observation of users programming. However, it was too “close to the textbook” to distil the most of each interview. Indeed, a well-prepared interview benefits from an original approach, so as to surprise oneself and avoid observing what was initially intended.

Moreover, I considered the method *think aloud* as an efficient way to gather users' needs, and applied it by instructing the participants to share their thoughts. It would perhaps have been more efficient to apply it meticulously instead, to bring the interviews off the beaten track and shed a different light on the problem, with the help of each participant.

References

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Prentice Hall.
- Bose, P. (1988). Interactive program improvement via EAVE: an expert adviser for vectorization. *Proceedings of the 2nd international conference on Supercomputing - ICS '88* (pp. 119–130). New York, New York, USA: ACM Press.
doi:10.1145/55364.55376
- Carroll, J., & Aaronson, A. (1988). Learning by doing with simulated intelligent help. *Communications of the ACM*, 31(9), 1064–1079. doi:10.1145/48529.48531
- De Bruijn, O., & Spence, R. (2008). A new framework for theory-based interaction design applied to serendipitous information retrieval. *ACM Transactions on Computer-Human Interaction*, 15(1), 5:1–5:38. doi:10.1145/1352782.1352787.
- Dow, S. P., Glassco, A., Kass, J., Schwarz, M., Schwartz, D. L., & Klemmer, S. R. (2010). Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. *ACM Transactions on Computer-Human Interaction*, 17(4), 1–24. doi:10.1145/1879831.1879836
- Kreeger, M. N. (2009). Security testing. *ACM SIGCSE Bulletin*, 41(2), 99–102.
doi:10.1145/1595453.1595484
- Lethbridge, T. C. (2000). What knowledge is important to a software professional? *Computer*, 33(5), 44–50. doi:10.1109/2.841783
- MathWorks. (n.d.). Using the MATLAB Code Analyzer Report. Retrieved September 7, 2012, from http://www.mathworks.se/help/techdoc/matlab_env/f9-11863.html
- Saffer, D. (2009). *Designing for Interaction: Creating Innovative Applications and Devices* (2nd ed.). New Riders Press.
- Sinha, R., & Swearingen, K. (2002). The role of transparency in recommender systems. *CHI '02 extended abstracts on Human factors in computer systems*, 830.
doi:10.1145/506558.506619
- Zhao, W., Jones, D. L., Cai, B., Whalley, D., Bailey, M. W., van Engelen, R., Yuan, X., et al. (2002). VISTA. *ACM SIGPLAN Notices*, 37(7), 155. doi:10.1145/566225.513857

Appendix A

This appendix contains the full discussion thread of the proposition submitted to GCC for a Google Summer of Code. The two first boxes are the initial answers to my proposal in Figure 9, my answer is split in the two last boxes.

While your idea sounds interesting, I think you are severely underestimating the complexity. The compiler usually performs thousands of transformations, not all of them in the same place and certainly not all of them in code that can tolerate injecting the kind of analysis code you will need to inject.

Additionally, not all the transformations the compiler does are easily mapped onto an expression or line of code that makes sense to the programmer (much of the code optimized by the compiler is a side-effect of other transformations or code implicitly generated to support higher-level abstractions).

Perhaps you could reduce the scope a bit by concentrating on a single pass. I would recommend looking at the vectorizer statistics output to get an idea.

I agree with [the previous reviewer]. But I would expect that most users are mostly interested in whether specific transformations were applied or not (e.g. “was my loop vectorized?”), so perhaps you could limit the scope of your project to a small number of major code transformations.

Alternatively, it would be a really useful project if you could instead make more passes use the statistics framework (and improve that framework while at it). This may not be as useful for users, but for compiler developers better organized information in the pass dumps would be very useful.

Hi,

The feedback needs actually not be fetched the closest to the transformation code, as long as we can report what GCC can and cannot do on a portion of code. Also, we do not need to be exhaustive on all the transformations done inside GCC. The intent of the feedback is to get the user browse the Internet to learn more on the standard optimization techniques. Ideally I would have put HTTP links to GCC docs in the feedback, but since it is impossible the optimization name is required to be cited.

A precision here for the reporting of “what GCC can and cannot do”. This is not meant as a list of optimizations which are actually implemented, and optimizations which will be in the future, for comparison with other compilers. This is rather a list of optimizations you can expect a compiler to perform, and portions of code which cannot be understood because of the impracticability of the expected transformation.

Placing the feedback calls closest to the transformations code had two main reasons:

- _ Reduced maintenance, as the transformation and the feedback are in the same source file.
- _ In a further improvement, each transformation unit could directly query the user (or a file representing his/her preferences) for hints, for example to choose the data structure underlying a map object. In fact we could also use the current feedback to propose the user to write a pragma hint.

My initial proposal involves coding only an `info_printf` for use by the maintainers of the various transformations. However, I was certainly optimistic with the scope of it, as we cannot expect the maintainers for each optimization to spontaneously update their code to output feedback. Most of them might actually not see any use for it, and will ignore it. If the final feedback includes only a handful of optimizations it will look like a useless proof of concept.

I can thus go for the single pass. If possible, I would rather still be using an `info_printf` function, so as to allow maintainers to further move it closer to the code, when mapping onto source code is possible of course.

The revised work plan:

- _ Enumerate all messages for each optimization technique, along with the observable GIMPLE/option flag pattern.
- _ Implement a function receiving feedback from each optimization unit and choosing whether to display it:
`info_printf(enum INFO_INDEX, const char*, ...);`
- _ Implement the messages for at least one optimization technique, through a single GIMPLE pass.
- _ Write a formatting guide for composing a feedback message.

There are still some very standard optimizations for which detection over GIMPLE is not trivial and would actually mean trying to make the job of the transformation itself. For example, CSE (*Common Subexpression Elimination*). In this case I would create a general message (i.e. not relating to source code), but still describing what triggers CSE. An other example is Dead code elimination, which depends on static profiling. There would be a message for the trivial `if (0)`, and a few general messages to introduce Value range propagation and its use to detect unreachable code.

So far I could list these optimizations which we could guess by overlooking on a single pass:

- _ SSA (*Single State Assignment*)
- _ Dead code elimination (for the `if (0)`)
- _ Strength reduction
- _ Function inlining
- _ Tail recursion
- _ Data alignment
- _ Stack optimizations
- _ `if` conversions (detection of simple conditional moves?)
- _ Register allocation (for the expectable storage of function arguments in registers – if there exists a common ground for the various ABIs – *Application Binary Interfaces*)
- _ Interprocedural analysis
- _ Math optimizations

For these other optimizations I will write a few general messages, and if possible they should eventually be moved closer to the transformation code:

- _ CSE
- _ Peephole/Superoptimization
- _ Vectorization
- _ Parallelization
- _ Instructions scheduling

My method for the GsoC (*Google Summer of Code*) would actually be listing the expectations from a user point of view rather than listing transformations as above. Here is how it would look like for Vectorization:

- _ What is vectorization? Which architectures?
- _ How can one ask for/verify Vectorization?
- _ Can GCC vectorize the same parallel arithmetic operation? How many at the same time?
- _ Can it vectorize a simple loop? How many minimal iterations?
- _ Can it detect/vectorize a [circular] shift?
- _ Can it understand my complex loop index?
- _ Can it manage my variable dependancies?
- _ How would n parallel additions in a loop be vectorized?

Here are the corresponding information bits which will form the feedback messages:

- _ Vectorization is SIMD (*Single Instruction Multiple Data*), performance boost, available on the main architectures [...], independent from the OS (*Operating System*) (General message – triggered when more than 2 arithmetic operations are performed, on a SIMD-capable architecture).
- _ Auto-vectorization enabled with `-O3, -ftree-vectorizer-verbose=2` to verify, difference between loop vectorization and SLP (*Superword Level Parallelism*) (General message – triggered if the previous message has already appeared).
- _ Arithmetic operations supported for vectorization, number in parallel depends on type and architecture, the number for "this" type on "this" architecture is [...].
- _ Certain to vectorize operations on range of array without cross-dependancies, overhead for vectorization and need for cost model, the loop threshold for "this" case is [...].
- _ ???
- _ linear indices understood in general, stride 1 or 2 actually vectorized, in "this" case it was/not vectorized
- _ Certain to manage dependencies with distance 1, involves following dependence path and handling cycles, try to avoid them
- _ ??? (Meant as a fun fact to show how loop vectorization and SLP interact)

Appendix B

This appendix contains two scanned interview sheets, respectively from the preparatory study and the user study with the prototypes.

Interviewee: Mikael

Date: 16/04/2012

This interview is part of my degree thesis, in which I am investigating the design of a feedback from the compiler to the user, to help refine the code and increase its quality. In short, it will be a discussion between you and the compiler in which it tries to make wise suggestions.

After a few preliminary questions, I will ask you to solve one or two typical short programming tasks, then I will ask you various questions on your experience as a programmer. The problem solving is not meant as a contest, but rather a "think aloud": I will need you to tell every single thought, interrogation, you had while solving the task. The overall interview will last at most 40 minutes.

- What is your main language of choice? Which tool/IDE do you use it with?

C++ mostly. many. Vim, GCC.

- How many years of programming do you have? In your main language?

11~12. 9~10.

- Thoughts during problem solving?

array bounds? cache locality, who like to learn. order of loop on indices. Compiler could prevent ~~from~~ where doing redundant calc. use of long long depending on bounds. /advise static string with enough size (arbitrary). like to keep things low level. would use typedef. abstract the types.

- What is usually your own target when programming? (examples: performance, portability, readability, maintenance, security, provability, simple design, extensibility, usability)

usab, extensibility for myself later.

- Do you think you can improve your current projects? Would it imply you improve your own capabilities? In the latter case, which resources would you browse? (examples: course, coding guideline, online tutorial, blog articles, books, hands-on experience)

yes, yes. Internet wandering. Small fixes.

$t[100][100]$, a, b, c

$$t[i][j] = \frac{i \times 2^a + j \times b}{c}$$

- country code

- name \rightarrow mail

- has shoe shop / rest / barber

- shoppers/year / range streets.

SWE

FRA

USA

Discussion: like debugger, free conversation. not too long. not automatic, I initialize.

Did not go so far as to wonder whether he can trust the compiler to apply an expected optim. in his place. If he knew an optim, would probably hand-code it.

Tester: YOUNES

Date: 30/08/12

The purpose of this work is to imagine how the compiler could give you feedback on its operation, to raise opportunities of improvements, and in general foster your mastery of the language.

I will show you three prototypes of development environments in which the compiler is giving such feedback, and ask you a few questions about them. They are not functional (you cannot code in them), the purpose is just to get the point.

➤ <http://www.csc.kth.se/~traf/thesis/proto1.html>

Read the code, what does it do? What is your bias about the compiler sending such random messages?
[positive / negative]

→ Prototype 1

Thoughts, possible improvements. Are the messages relevant? Understandable? Should they be more complex (thus longer to read) or not? Would you prefer links to technical sources spread on the web, or to a single concise documentation?

No use for[#], prime numbers. Links good. Ok if too complex, always
3rd interesting, tells why & I learned sth.
Not really relevant, but no need to disable it like Reddit :)

Prototype 2

Thoughts, possible improvements. There are two aspects here: you can click the elements to tune them, or the compiler can propose you to tune the most influential ones. Which would you favor?

Helpful. There are tools already avail, good to integrate. Oxide
Good that no learning. Don't know what can discussion accomplish.

Prototype 3

Thoughts. Should the information be more detailed / summarized? How do you interpret the color code?

Looks puzzled. ~~Does~~ cares?
Does not get what's right or wrong.
Does crazy coding, this proto would be annoying to show what I know!
(How would it have been if in other order protos?)

Personas

If I tell you this recommendation was written by the creator of Linux, does it make it more valuable? The idea is to have influent people create these messages, and let you choose who you would trust. Would you think about someone in particular?

Great to learn from someone
Contribute to community instead of friends.

TRITA-CSC-E 2012:084
ISRN-KTH/CSC/E--12/084-SE
ISSN-1653-5715