Fernando Magno Quintão Pereira (Ed.)

LNCS 8771

Programming Languages

18th Brazilian Symposium, SBLP 2014
Maceio, Brazil, October 2–3, 2014
Proceedings

Springer

Lecture Notes in Computer Science 8771

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Fernando Magno Quintão Pereira (Ed.)

Programming Languages

18th Brazilian Symposium, SBLP 2014
Maceio, Brazil, October 2-3, 2014
Proceedings

Springer

Volume Editor

Fernando Magno Quintão Pereira
Federal University of Minas Gerais
Av. Antônio Carlos
Belo Horizonte, Minas Gerais, Brazil
E-mail: fernando@dcc.ufmg.br

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-319-11862-8 e-ISBN 978-3-319-11863-5
DOI 10.1007/978-3-319-11863-5
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014949227

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of the 18th Brazilian Symposium on Programing Languages (SBLP 2014), held during October 2nd/3rd of 2014, in Maceió, Brazil. The Brazilian Symposium on Programing Languages had its first event in 1996, and since 2010, it has been part of the Brazilian Conference on Software (CBSoft). This symposium is a venue where researchers, developers, educators, and practitioners exchange information on both the theory and practice related to all the aspects of programing languages and systems.

Interest in the symposium has been confirmed by the submission of papers on the most diverse subjects. There were 31 submissions from authors from six different countries. Each submitted paper was reviewed by at least three members of the Program Committee. The expert opinions of many outside reviewers were invaluable to ensure the high quality of the program. We express our sincere gratitude for the effort of these volunteers. The final program featured two invited talks, two tutorials, eleven full papers in English, and three papers in Portuguese. These three papers, which were presented at the conference, are not part of these proceedings.

We owe a great deal of thanks to the authors, reviewers, and the members of the Program Committee for making the 18th SBLP a success. We thank Louis-Noel Pouchet for his invited talk on optimizing compilers for high-performance computing, and we thank Fabrice Rastello for his talk on the duality that exists between static and dynamic program analyses. Finally, we thank Márcio Ribeiro, Leandro Dias da Silva and Baldoíno Fonseca, the general organizers of CBSoft 2014 for all their help and support.

August 2014 Fernando Magno Quintão Pereira

Organization

Program Chair

Fernando Pereira	UFMG, Brazil

Local Arrangement (co)-Chair(s)

Baldoíno Fonseca	UFAL, Brazil
Márcio Ribeiro	UFAL, Brazil
Leandro Dias da Silva	UFAL, Brazil

Program Committee

Alberto Pardo	Universidad de la República, Uruguay
Alex Garcia	IME, Brazil
Álvaro Moreira	Federal University of Rio Grande do Sul, Brazil
Ándre Rauber Du Bois	Federal University of Pelotas, Brazil
Carlos Camarão	Federal University of Minas Gerais, Brazil
Christiano Braga	Fluminense Federal University, Brazil
Fábio Mascarenhas	Federal University of Rio de Janeiro, Brazil
Fabrice Rastello	Inria, France
Fernando Pereira	Federal University of Minas Gerais, Brazil
Fernando Castor	Federal University of Pernambuco, Brazil
Francisco Carvalho-Júnior	Federal University of Ceará, Brazil
Hans-Wolfgang Loidl	Heriot-Watt University, UK
João Saraiva	University of Minho, Portugal
João F. Ferreira	Teesside University, UK
Louis-Noel Pouchet	University of California Los Angeles/Ohio State University, USA
Lucilía Figueiredo	Federal University of Ouro Preto, Brazil
Luis Barbosa	University of Minho, Portugal
Manuel A. Martins	University of Aveiro, Portugal
Marcello Bonsangue	Leiden University, The Netherlands
Marcelo Maia	Federal University of Uberlândia, Brazil
Marcelo D'Amorim	Federal University of Pernambuco, Brazil

Mariza Bigonha	Federal University of Minas Gerais, Brazil
Martin Musicante	Federal University of Rio Grande do Norte, Brazil
Noemi Rodriguez	PUC-Rio, Brazil
Peter Mosses	Swansea University, UK
Rafael Lins	Federal University of Pernambuco, Brazil
Renato Cerqueira	PUC-Rio, Brazil
Roberto Bigonha	Federal University of Minas Gerais, Brazil
Rodrigo Geraldo	Federal University of Ouro Preto, Brazil
Sandro Rigo	State University of Campinas, Brazil
Sergio Medeiros	Federal University of Rio Grande do Norte, Brazil
Simon Thompson	University of Kent, UK
Varmo Vene	University of Tartu, Estonia

Invited Reviewers

Cardoso, Elton	Moraes, Bruno	Rodrigues, Bruno
Cunha, Jácome	Neves, Renato	Souza Neto, Plácido
Madeira, Alexandre	Ribeiro, Rodrigo	Voidani, Vesal

Table of Contents

A Mixed Approach for Building Extensible Parsers

Leonardo Vieira dos Santos Reis[1], Vladimir Oliveira Di Iorio[2], and Roberto S. Bigonha[3]

[1] Departamento de Computação e Sistemas, Universidade Federal de Ouro Preto, João Monlevade, Brazil
leo@decsi.ufop.br
[2] Departamento de Informática, Universidade Federal de Viçosa, Viçosa, Brazil
vladimir@dpi.ufv.br
[3] Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
bigonha@dcc.ufmg.br

Abstract. For languages whose syntax is fixed, parsers are usually built with a static structure. The implementation of features like macro mechanisms or extensible languages requires the use of parsers that may be dynamically extended. In this work, we discuss a mixed approach for building efficient top-down dynamically extensible parsers. Our view is based on the fact that a large part of the parser code can be statically compiled and only the parts that are dynamic should be interpreted for a more efficient processing. We propose the generation of code for the base parser, in which hooks are included to allow efficient extension of the underlying grammar and activation of a grammar interpreter whenever it is necessary to use an extended syntax. As a proof of concept, we present a prototype implementation of a parser generator using Adaptable Parsing Expression Grammars (APEG) as the underlying method for syntax definition. We show that APEG has features which allow an efficient implementation using the proposed mixed approach.

1 Introduction

Parser generators have been used for more than 50 years. Tools like YACC [10] can automatically build a parser from a formal definition of the syntax of a language, usually based on context-free grammars (CFG). The main motivation for automatic parser generation is compiler correctness and recognition completeness, since with manual implementation it is very difficult to guarantee that all programs in a given language will be correctly analysed. The parsers that are generated from the language formal definition usually consist of a fixed code driven by a parse table. Different languages are associated with distinct parse tables. With top-down parsers, it is common to produce the code directly as a recursive descent program instead of using parse tables.

Extensible parsers [28] impose new challenges to automatic parser generation, since they have to cope with on-the-fly extensions of their own concrete syntax,

F.M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 1–15, 2014.

requiring that the parser must be dynamically updated. If these changes are frequent, the use of interpretation techniques may be less time-consuming than to reconstruct the parser. Thus, at first, instead of building a fixed code for the parser, it may be interesting to use an interpreter for parsing the input based on a suitable representation of the syntax of the language, which may be dynamically updated. However, this solution is not entirely satisfactory. A mixed approach, compilation and interpretation, offers the best of both worlds.

For automatically building a parser, either using code generation or interpretation, it is necessary to use a formal model that is powerful enough to appropriately describe the syntax of the language, including possible on-the-fly modifications. Adaptable Parser Expression Grammar (APEG) [17,18] is a new formal model that satisfies these requirements. It is based on PEG (Parsing Expressions Grammars) [7], a formalism similar to CFG which formally describes a recursive descent parser. APEG extends PEG with the notion of *adaptability*, which is implemented by means of operations that allow the own syntax of the language to be extended. In previous work [19], we have shown that a prototype interpreter for APEG allows the implementation of extensible parsers which are more efficient than the traditional approaches used by other tools.

In this work, we discuss a mixed approach for building extensible parsers. Our view is based on the fact that a large part of the syntax of an extensible language is stable, so it is appropriate for a parser whose code can be statically generated. The parts of the syntax specification that are dynamically extended may use grammar interpretation for a more efficient processing. We propose the generation of code for the base syntax, including hooks that may allow an efficient extension, activating an interpreter whenever it is necessary to use the extended syntax. As a proof of concept, we present a prototype implementation of a parser generator for APEG. We show that APEG has features which allow an efficient implementation using the proposed mixed approach.

Section 2 presents a definition of the APEG model to help understanding the concepts used in the following sections. Section 3 discusses our approach in detail, showing how it works with the APEG model. In Section 4, we describe some efficiency tests with parsers generated using the mixed approach. Section 5 lists some works similar to ours. Section 6 presents our final conclusions and discusses future works.

2 Adaptable Parsing Expression Grammar

Parsing Expression Grammar (PEG) [7] is a model for describing the syntax of programming languages using a recognition-based approach instead of the generative system typical of context-free grammars (CFG). Similar to CFG, formally a PEG is a 4-tuple (V_N, V_T, R, S), where V_N is the set of nonterminal symbols, V_T is the set of terminal symbols and S is the initial symbol. R is a rule function which maps nonterminals to *parsing expressions*. A parsing expression is very similar to the right hand side of a CFG production rule in the extended Backus-Naur form, except for the addition of two new operators: the *not-predicate* and the *prioritized choice*. The not-predicate operator

checks whether the input string matches some syntax, without consuming it, thus allowing unrestricted lookahead. The prioritized choice lists alternative patterns to be tested in order.

Adaptable PEG (APEG) [18] is an adaptable model based on PEG. It also uses the idea of attributes of Attributes Grammars [27] to guide parsing. The attributes of APEG are L-Attributed and its purporse is syntactic and not semantics as in attributes grammars. APEG possesses a special attribute called *language attribute*, which represents the set of rules that are currently used. Language attribute values can be defined by means of embedded semantic actions and can be passed to different branches of the parse tree. This allows a formal representation of a set of syntactic rules that can be modified on-the-fly, i.e., during the parsing of an input string. APEG allows to extend the grammar by addition of new choices at the end of an existing list of choices of the definition rule of a given nonterminal or by addition of new nonterminal definitions.

As a concrete example, Figure 1 shows a PEG definition of a toy block structured language in which a block consists of a list of declarations of integer variables, followed by a list of assignment statements. An assignment statement consists of a variable on the left side and a variable on the right side. For simplicity, the whitespaces are not considered.

$block \leftarrow \{\ dlist\ slist\ \}$	$decl \leftarrow \textbf{int}\ id\ ;$
$dlist \leftarrow decl\ decl^*$	$stmt \leftarrow id = id\ ;$
$slist \leftarrow stmt\ stmt^*$	$id \leftarrow alpha\ alpha^*$

Fig. 1. Syntax of block with declaration and use of variables (simplified)

Suppose that the context dependent constraints of this language are: a variable cannot be used if it has not been declared before, and a variable cannot be declared more than once. Using APEG, we implemented these context dependent constraints as shown in Figure 2. As mentioned before, every nonterminal has a special inherited attribute, the language attribute, which is a representation of a grammar. In Figure 2, this attribute is always the first one and it has type Grammar. The inherited attributes of a nonterminal are enclosed in the symbols "[" and "]" occurring after the name of the nonterminal, and the synthesized attributes are specified after the *returns* keywords. For example, line 20 defines the nonterminal *id* with one inherited attribute of type *Grammar* named as *g* (it is the language attribute) and one synthesized attribute named as *s* of type *String*. To refer to a nonterminal on a parsing expression, we use the name of the nonterminal followed by the list of inherited attributes and synthesized attributes, in this order, enclosed in the symbols "<" and ">". As an example, in line 21, *alpha<g, ch1>* refers to the nonterminal *alpha* followed by its attributes *g* and *ch1*. APEG also has *constraint*, *binding* and *update expressions*. The *constraint expression* is a boolean expression enclosed by the symbols "{?" and "}", such as the expression in the definition of the nonterminal *var* (line 12). A *constraint expression* succeeds if it evaluates to true, otherwise it fails. A *binding*

expression assigns the input string matched by a parsing expression to a variable. It is used on line 24 of Figure 2 to assign to variable *ch* the value of the character matched by the given parsing expression. An *update expression* is enclosed by the symbols "{" and "}" and it is used to assign the value of an expression to an attribute. The parsing expression $\{g = g1;\}$ in line 5 of Figure 2 is an example of an update expression.

In this example, the idea of implementing the context dependent constraints is to adapt the nonterminal *var* on the fly in order to allow only declared variables to be recognized. Note that, in the beginning, the nonterminal *var* does not recognize any symbols (lines 11-12). However, when a variable is declared (nonterminal *decl*, defined in lines 7-9), a new grammar rule is produced by the addition of a new choice in the definition of nonterminal *var*, which allows the recognition of the new variable name. The resulting new grammar is passed as the language attribute, in the definition of the nonterminal block, to the nonterminal *slist*, and, in the sequel, to *stmt*. As a result, the nonterminal *stmt* now can recognize the declared variable.

```
1  block[Grammar g]:
2    '{' dlist<g, g1> slist<g1> '}' !.;
3
4  dlist[Grammar g] returns[Grammar g1]:
5    decl<g, g1> {g = g1;} (decl<g, g1> {g = g1;})*;
6
7  decl[Grammar g] returns[Grammar g1]:
8    !('int␣' var) 'int␣' id<s> ';'
9     {g1 = g + 'var:␣\'' + s + '\' !alpha<g, ch>;';};
10
11 var[Grammar g]:
12   {? false };
13
14 slist[Grammar g]:
15   stmt<g> stmt<g>*;
16
17 stmt[Grammar g]:
18   var<g> '=' var<g> ';';
19
20 id[Grammar g] returns[String s]:
21   alpha<g, ch1> {s = ch1;} (alpha<g, ch2> {s = s + ch2;})*;
22
23 alpha[Grammar g] returns[String ch]:
24   ch=[a-zA-Z0-9_];
```

Fig. 2. Example with the set of production rules changed at parse time

For example, suppose the input string {*int a;int b;a=b;b=a;*}. The recognition of this string starts with the nonterminal *block* and its language attribute is the grammar in Figure 2. After recognizing the first symbol, {, the parser proceeds to recognize a list of declarations (nonterminal *dlist*), passing down the same grammar as the language attribute to the nonterminal *dlist*. During the recognition of the nonterminal *dlist*, it first tries to match a variable declaration through the nonterminal *decl*, passing to it the same language attribute. The nonterminal *decl* first checks if the variable is already declared using the parsing expression *!('int ' var)*. The not-lookahead operator, !, succeeds if the expression enclosed in parentheses fails, and it does not consume any symbol from the input. In order to check whether the variable "a" has already been declared, the parsing expression enclosed in parentheses matches the "int" string, but the nonterminal *var* does not recognize the variable "a", because it does not have any rule for it yet. In the sequel, the parsing expression *'int ' id⟨s⟩';'* recognizes the declaration of variable 'a'. Note the use of the nonterminal *id* instead of the nonterminal *var*. The nonterminal *id* is used here to recognize any valid variable name, since it is a new one. Then, a new grammar is built from the current grammar by the addition of a new choice, *var* : 'a' !*alpha⟨ch⟩*;, on the definition of nonterminal *var*. This new grammar becomes the value of the synthesized attribute $g1$.

The grammar synthesized by the nonterminal *decl* is used in the nonterminal *dlist* as language attribute of other calls of the nonterminal *decl*. Proceeding, the next variable declaration will be recognized, and the nonterminal *dlist* synthesizes a new grammar with these two choices, in this order, *var* : 'a' !*alpha⟨ch⟩*; and *var* : 'b' !*alpha⟨ch⟩*;, for the nonterminal *var*. This new grammar is used by the nonterminal *block* to pass it as the language attribute of the nonterminal *slist*. As a result, the two statements, $a = b$ and $b = a$, can be recognized, because the nonterminal *var* in the language attribute passed to the nonterminal *stmt* has rules to recognize the variables 'a' and 'b'.

Usually, parser generators for PEG produce a top-down recursive descent parser. Every nonterminal is implemented by a function whose body is a code for its parsing expression. The return value of each function is an integer value representing the position on the input that it has got moved on or the value -1 if it fails. It is straightforward to extend this idea to include attributes: the inherited attributes become parameters of functions and synthesized attributes are return values. For example, Figure 3 shows the code generated for the nonterminal *var* of Figure 2. The function *var* has one parameter, the language attribute, and returns an object of type *Result*, which must contain fields representing the portion of the input consumed and the values of the synthesized attributes, when specified.

Complications with this scheme arise when the base grammar is dynamically extended during parsing. When new choices are added to the *var* nonterminal, the function of Figure 3 does not represent anymore the correct code for this nonterminal, then this function must be updated. However, it is cumbersome regenerating all the parser code on the fly to reflect these small changes. In

```
1 Result var(Grammar g) {
2   if(false) {
3       // do nothing
4   } else
5     return new Result(-1); // a fail result
6 }
```

Fig. 3. Example of code generated by a PEG

cases where the grammar changes several times, as in extensible languages, the on-the-fly regeneration of all the parser is very expensive [17]. An alternative solution is to interpret the whole grammar directly. However, this may cause a great loss in parsing efficiency. So, we propose, in this paper, an approach to efficiently adapt the grammar. We propose to generate the code from the base grammar and include hooks to jump from the generated code to interprete the parts that have been added dynamically.

3 Mixing Code Generation and Interpretation

Since APEG only allows changes in the definition of nonterminal symbols by insertion of new choices at the end of the rules [19], we generate a recursive descent parser from an APEG grammar, so there is a function for each nonterminal and, whenever necessary, we place at the end of the body of these functions a call to the interpreter.

Figure 4 shows a scratch of the code generated for the grammar of Figure 2. As shown, we generate a Java class which has a function to each nonterminal definition on the grammar. The generated class extends the predefined class *Grammar* that has the implementation of standard functions, such as the function *interpretChoice* to interpret an AST and functions to add rules to the grammar or to clone the grammar itself.

The vector *adapt* (line 3 in Figure 4) stores a possible new choice for each nonterminal. Notice the hook at the end of the function body of each nonterminal (lines 29 and 40) to call the interpreter with its possible choice. This hook will be reached only if its preceding code fails, indicating that we must interpret the new choice. For example, if the code representing the parsing expression *'{ 'dlist<g, g1> slist<g1> '} '* on lines 6 to 24 fails, then we call the interpreter passing its new choice. So, the action of adapting a grammar is just an action of including a new choice rule on the vector *adapt*.

Using this strategy, all the base code for the grammar is generated and compiled, and only the choices that are added dynamically must be interpreted. Our strategy is based on the assumption that the code for the base grammar is expected to be large and used many times. Therefore, the expected result shall be a faster parser than the interpreter we have proposed in previous work [19], and will still allow an efficient method for managing syntactic extensions.

```
1  public class BlockLanguage extends Grammar {
2    // vector of new choices
3    private CommonTree[] adapt = new CommonTree[8];
4    ...
5    public Result block(BlockLanguage g) {
6      BlockLanguage g1; // local attribute
7      Result result;
8      int position = g.match("{", currentPos);
9      if(position > 0) {
10       g.currentPos = position;
11       result = g.dlist(g);
12       if(!result.isFail()) {
13         g1 = (BlockLanguage) result.getAttribute(0);
14         g1.currentPos = result.getNext_pos();
15         result = g1.slist(g1);
16         if(!result.isFail()) {
17           position = g.match("}", result.getNext_pos());
18           if(position > 0) {
19             char ch = g.read(position);
20             if(APEGInputStream.isEOF(ch))
21               return new Result(position);
22           }
23         }
24       }
25     }
26     Environment env;
27     ... // set the environment to start the interpreter
28
29     // interpreter the choice of block (index 0)
30     return g.interpretChoice(adapt[0], env);
31   }
32
33   public Result var(BlockLanguage g) {
34     if(false) {
35       // do nothing
36     }
37     Environment env;
38     // set the environment to start the interpreter
39
40     // interpreter the choice of var (index 3)
41      return g.interpretChoice(adapt[3], env);
42   }
43   ... // other functions
44 }
```

Fig. 4. Generated code for the block language

In the APEG model, a parsing expression of a nonterminal is fetched from its language attribute. Using different language attributes, it is possible to get different parsing expressions for the same nonterminal, thus effectively adapting the grammar. To have this behaviour, each function generated from a nonterminal has the language attribute as a parameter. The type of this parameter is the type of the grammar generated. In our example, Figure 4 shows the language attribute, whose type is *BlockLanguage*, of the functions *block* (line 5) and *var* (line 32).

We use the *dot* notation to call a nonterminal function associated with its correct language attribute. For example, the nonterminals *dlist* and *slist* on the definition of the function *block* are called as *g.dlist(g)* (line 11) and *g1.slist(g1)* (line 15). Note that, as the language attribute passed to each nonterminal is different, we call each nonterminal function from a different language attribute. We must call *slist* from the object *g1* instead of *g* because the vector *adapt* of *g1* has a different value of choices for the function *var*. So, the interpreter is called, the new choice is passed, allowing the use of the variables that have been declared.

A restriction to this approach is that as we use the generated class as the language attribute type, e.g. the *BlockLanguage* type in Figure 4, it is not possible to pass a different grammar which is not subtype of the generated class, as the language attribute. For example, suppose a grammar with other definitions for the same nonterminals presented in Figure 4. One may want to pass as the language attribute this grammar in a specific context on the definition of the block language of Figure 4. However, as this grammar is not a subclass of *BlockLanguage*, there will be a type error. Instead of using the generated class as the language attribute, we could use the base type, *Grammar*, as the language attribute and use reflection on runtime to invoke the nonterminal functions. However, as the use of reflection may result in a slower program than the use of the dot notation to call functions, we avoid this solution.

During the interpretation process of a parsing expression, it is possible to encounter a reference to a predefined nonterminal. In this case, the interpreter must execute the function code of this nonterminal. For example, suppose an input to the block language example of Figure 2 which adds the choice *var* : 'a' !*alpha*⟨*ch*⟩; to the definition of the nonterminal *var*. The nonterminal referenced in this choice, *alpha*, is the one defined in Figure 2 and has a code generated for it. So, when the interpreter reaches this nonterminal, it must stop interpreting and invoke the function of this nonterminal. We implemented this feature using the reflection mechanism of the Java language. Whenever interpreting a nonterminal, the interpreter checks whether the nonterminal is a method of the language attribute object, and if so, the interpreter invokes the method code by reflection. Otherwise, it continues the interpretation.

The code presented in Figure 4 was not automatically generated. In order to test our approach, we first produced handwritten code for some APEG specifications, such as the one presented in Figure 2 and another one discussed in Section 4. For the interpretation, we modified a prototype interpreter we had

developed in a previous work [19]. One of the main modifications was the code for calling, from the interpreter, functions on the generated code. This feature was implemented using reflection in the Java language, as previously discussed. After our mixed approach proposal be proved useful, we will write the code generator to automatically produce a recursive descent parser from an APEG specification.

4 Evaluation of the Mixed Extensible Parser

We have performed preliminary experimental tests to evaluate whether our mixed approach is feasible. We were interested in the performance of the mixed code, i.e., the cost of switching to the interpreter and turning back to the compiled code. So, we built tests which exercises these features. We used two language definitions to evaluate our approach: the block language presented in Section 2 and a version of a data dependent language presented in [9]. The syntax of the data dependent language is an integer followed by the same number of characters enclosed by the symbols "[" and "]". The input *3[abc]* is an example of valid string of this language. Figure 5 shows an APEG grammar for this language. Note that it adds a new choice to the nonterminal *strN* with exactly the number of characters given by the integer value just read, e.g, for the input *3[abc]*, the nonterminal *strN* is extended with the rule $strN \rightarrow CHAR\ CHAR\ CHAR$.

```
1  literal[Grammar g]:
2    number<n>
3    { g1 = g + 'strN␣:␣' + concatN('CHAR␣', n) + ';'; }
4    '[' strN<g1> ']'
5  ;
6
7  strN[Grammar g]: {? false } ;
8
9  number returns[int r]: t=[0-9]+ { r = strToInt(t); } ;
10
11 CHAR : . ;
```

Fig. 5. APEG grammar for a data dependent language

We used these APEG grammars because they are simple and demand switching between the compiler and the interpreter. The data dependent example will adapt the grammar once and force the interpreter to return to the code of the *CHAR* function many times. The block language example adapts the grammar several times and also turns back from the interpreter to the compiler code every time a variable is used or declared. We have performed the experiments in a 64-bit, 2.4 GHz Intel Core i5 running Ubuntu 12.04 with 6 GB of RAM on

the Eclipse environment. We have repeated the execution 10 times in a row and computed the average execution time.

Table 1 shows the result for the data dependent language. The inputs used were automatically generated by setting an integer number and then randomically generating this set of characters. The first column shows the value of the integer used in the input string; in this case, the size of the input is proportional to this value. The second column shows the time for parsing the input string, using a prototype interpreter we have developed in a previous work. We have shown that this interpreter presents better performance than similar works, when used for languages requiring extensibility [19]. The third column presents the time for parsing the same input string using our new approach, mixing code generation and interpretation. This result shows that, even though using reflection to switch between interpreter and compiled code is expensive, the efficiency of the compiled code compensates it.

Table 1. Time in milliseconds for parsing data dependent programs. The performance of the interpreter and the mixed approach are compared.

Size	Interpreter	Mixed approach
1000	258	276
10000	1615	2584
100000	68744	36015
150000	181338	86576
200000	445036	164041

Table 2 shows the results of parsing programs of the block structure language of Figure 2. The first column shows the number of variables declared and the second column shows the number of assignment statements in the programs used as input string. These programs were also automatically generated by creating a set of variables and formed assignment statement by chosing two variables of this set. The third and the fourth columns present the time for parsing the programs using the prototype interpreter and using a mixed approach, respectively. The results show that the mixed approach executes slightly faster than the interpreter.

Table 2. Time in milliseconds for parsing block language programs. The performance of the interpreter and the mixed approach are compared.

Variable	Statements	Interpreter	Mixed approach
100	1	176	176
500	1	754	498
1000	1	912	725
100	100	232	263
100	500	448	295
100	1000	506	394

These examples force the use of the slow mechanism of reflection several times. In real cases, we expect the activation of interpretation and reflection is not too frequent, thus the performance of the parser using the mixed approach would be even better. So, our preliminary experiments indicate that the mixed approach can really improve the APEG performance.

5 Related Work

In 1971, based on data from empirical studies, Knuth [12] observed that most programs spend the majority of time executing a minority of code. Using these observations, Dakin and Poole [4] and Dawson [5] (both works published in a same journal issue) independently proposed the idea of "mixed code". The term refers to the implementation of a program as a mixture of compiled native code, generated for the frequently executed parts of the program, and interpreted code, for the less frequently executed parts of the program. Their main motivation was to save memory space with little impact on the execution speed of programs. The interpreted source code is usually smaller although slower than the generated native code. Although with different motivations, this approach has some similarities to our work in the sense that the native code is statically generated (in our case, code for a large part of the parser is statically generated) and interpretation is also used.

Plezbert [15] proposes the use of a mixture of compiled and interpreted code in order to improve programming efficiency during software development. His purpose is to reduce the time spent in the "make" process, considering that programmers repeatedly use the cycle edit-make-execute when developing software. Here, "make" stands for the compilation of files that have been changed, compilation of files dependent on the ones which have been changed, and linking of separately compiled objects into an executable image. The increasingly use of aggressive compiler optimizations causes the make process to take longer. A solution may be the use of interpretation for prototyping, during the development phase, or to turn off optimizations until a final release is to be produced. Plezbert suggests an alternative approach which he calls "continuous compilation". After editing a program, the execution phase can immediately start using interpretation, while the "make" phase is performed concurrently with program execution. The interpreted code is gradually replaced by natively-executable code. Performance increases until, eventually, the entire program has been translated to a fully optimized native form. In our work, extensions for the parser are always interpreted. We could benefit from the "continuous compilation" approach if code is concurrently generated for extensions, without stopping the parsing process. When the code is completely generated and compiled, it could replace the interpreted parts of the parser. Further investigation is necessary, but we believe the opportunities for the performance gains in parsing are smaller then the ones reported by Plezbert.

Just-in-time compilation (JIT) [1], also known as *dynamic translation*, is compilation done during execution of a program – at run time – rather than prior to

execution. JIT is a means to improve the time and space efficiency of programs, using the benefits of compilation (compiled programs run faster) and interpretation (interpreted programs are typically smaller, tend to be more portable and have more access to runtime information). Thompson's paper [23] is frequently cited as one of the first works to use techniques that can be considered JIT compilation, translating regular expressions into IBM 7094 code.

The success of Java has increased the attention to JIT compilation, highlighting the tradeoff between portability of code and efficiency of execution. Several Java implementations have been developed using JIT, such as the ones provided by Sun [3], IBM [22] and the Harissa environment [13,14]. Several improvements to JIT have been proposed, extending the possibilities of mixing interpretation and compilation. For example, Plezbert has shown that good results could be achieved combining JIT compilation with his approach of "continuous compilation", which he called "smart JIT" [16]. Dong Heon Jung et alli [11] add to JIT the approaches of "ahead-of-time compilation" and "idle-time compilation", building a hybrid environment in order to increase the efficiency on a Java-based digital TV platform. It should be noted that our approach is the reversal of the traditional JIT in the sense that we perform a "just-in-time interpretation" during the execution of a compiled code. However, we share the objective of improving execution speed.

All the works discussed so far in this section use a combination of interpretation and code generation, having goals such as performance gain and portability, but not specifically involving parsing. The works discussed in the following are related to improvements on the parsing process, when extensions are considered.

Parsers using a bottom-up approach are usually built as a small, fixed code that is driven by a large parse table, generated from a formal specification of a language. When an extension to the language is required, the entire parse table must be recalculated, which is an expensive process. Several works propose techniques for generating small parse tables for the extended parts of the language, and combining them with the table originally generated for the base language. As an example, Schwerdfeger and Van Wyk [21,20] define conditions for composing parsing tables while guaranteeing deterministic parsing. The algorithm described by Bravenboer and Visser [2] for parse table composition supports separate compilation of grammars to parse table components, using modular definition of syntax. A prototype for this algorithm generates parse tables for scannerless Generalized LR (GLR) parsers [24], with input grammars defined in SDF [25]. The use of GLR imposes no restrictions on the input grammars, allowing a more natural definition for the syntax than methods based on PEG, such as our approach. On the other hand, GLR does not guarantee linear time processing for the generated parsers. In [17], we have shown that a prototype interpreter using APEG may present better performance results than works based on GLR, when dynamic parser extensions are required, even not considering yet the improvements provided by mixing interpretation with code generation. The obvious reason is that these works are not designed having dynamic extensibility

as an important goal. A disadvantage of the current version of APEG is that it does not offer facilities for modular specifications.

OMeta [26] is a fully dynamic language extension tool, allowing lexically scoped syntax extensions. Similarly to our work, it is based in Parsing Expression Grammar, but it can make use of a number of extensions in order to handle arbitrary kinds of data (not limited to processing streams of characters). Also similarly to our work, OMeta extends PEG with semantic predicates and semantic actions, which can be written using the host language. Programmers may create syntax extensions by writing one or more OMeta productions inside {}s. This creates a new parser object (at parsing time) that inherits from the current parser. In current implementations, everything is processed during parsing time, so they have more in common with our previous work, using only interpretation.

Hansen [8] designed a dynamically extensible parser library and two new algorithms: an algorithm for incremental generation of LL parsers and a practical algorithm for generalized breadthfirst top-down parsing. This work is similar to ours in the sense that the parsers produced may be modified on-the-fly, during parsing time, and it also uses top-down parsing methods. Although the algorithms proposed by Hansen have an exponential worst-case time complexity, the author showed that they may work well in practice. Our approach is based on PEG, so it always produces parsers with linear-time processing, provided by the use of memoization. It may be interesting to implement the examples used by Hansen (a Java grammar and several extensions) in APEG and compare the performance of the two approaches, for parser generation and for parser execution.

6 Conclusion and Future Work

Automatic generation of an extensible parser is difficult because extensions on the syntax may invalidate the generated parser code. In order to ameliorate this problem, in this paper, we propose a novel mixed approach to generate an extensible parser, which combines compilation with interpretation, using Adaptable Parsing Expression Grammars (APEG) as the underlying formal model. The greatest virtue of this proposal is its simplicity, which comes from the APEG model.

Preliminary experiments indicate that the mixed approach can improve the performance of the APEG parser. Our goals in the experiments were to evaluate the mixed code, thus we used languages and examples that exercise this feature. The next steps are to evaluate the performance of the mixed extensible parser in real situations, such as parsing programs of extensible languages like SugarJ [6]. Our strategy is based on the assumption that the code for the base grammar is expected to be large and used many times than the extensions. We still have to prove this assumption.

As APEG allows changing the grammar during the parsing, it opens several possibilities to compose grammars, and in the field of grammar extensibility, allowing, for example, simulate a kind of backbones described in [9]. We plan to investigate and evaluate the performance of the parser in these situations.

As explained in Section 3, we have not developed yet a code generator that may automatically produce a recursive descent parser for the static part of the language specification written in APEG. The examples used in this paper were handwritten and served only for a preliminary tests of the proposed approach. Our next steps include the implementation of this code generator, which will make possible to test for the entire syntax of real extensible languages. Several optimizations on the generated parsers may also be introduced. For example, we can apply techniques to generate code for rules that are used more frequently during interpretation.

References

1. Aycock, J.: A brief history of just-in-time. ACM Comput. Surv. 35(2), 97–113 (2003)
2. Bravenboer, M., Visser, E.: Parse Table Composition – Separate Compilation and Binary Extensibility of Grammars. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 74–94. Springer, Heidelberg (2009)
3. Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., Wolczko, M.: Compiling java just in time. IEEE Micro 17(3), 36–43 (1997)
4. Dakin, R.J., Poole, P.C.: A mixed code approach. Comput. J. 16(3), 219–222 (1973)
5. Dawson, J.L.: Combining interpretive code with machine code. Comput. J. 16(3), 216–219 (1973)
6. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: Sugarj: library-based syntactic language extensibility. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2011, pp. 391–406. ACM, New York (2011)
7. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. SIGPLAN Not. 39(1), 111–122 (2004)
8. Hansen, C.P.: An Efficient, Dynamically Extensible ELL Parser Library. Master's thesis, Aarhus Universitet (2004)
9. Jim, T., Mandelbaum, Y., Walker, D.: Semantics and algorithms for data-dependent grammars. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 417–430. ACM, New York (2010)
10. Johnson, S.C.: Yacc: Yet Another Compiler Compiler. In: UNIX Programmer's Manual, vol. 2, pp. 353–387. Holt, Rinehart, and Winston, New York (1979)
11. Jung, D.-H., Moon, S.-M., Oh, H.-S.: Hybrid compilation and optimization for java-based digital tv platforms. ACM Trans. Embed. Comput. Syst. 13(2s), 62:1–62:27 (2014)
12. Knuth, D.E.: An empirical study of fortran programs. Software – Practice and Experience 1(2), 105–133 (1971)
13. Muller, G., Moura, B., Bellard, F., Consel, C.: Harissa: A flexible and efficient java environment mixing bytecode and compiled code. In: Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS 1997), vol. 3, p. 1. USENIX Association, Berkeley (1997)
14. Muller, G., Schultz, U.P.: Harissa: A hybrid approach to java execution. IEEE Softw. 16(2), 44–51 (1999)
15. Plezbert, M.: Continuous Compilation for Software Development and Mobile Computing. Master's thesis, Washington University, Saint Louis, Missouri (1996)

16. Plezbert, M.P., Cytron, R.K.: Does "just in time" = "better late than never"? In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997, pp. 120–131. ACM, New York (1997)
17. Reis, L.V.S., Di Iorio, V.O., Bigonha, R.S.: Defining the syntax of extensible languages. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC 2014, pp. 1570–1576 (2014)
18. dos Santos Reis, L.V., da Silva Bigonha, R., Di Iorio, V.O., de Souza Amorim, L.E.: Adaptable parsing expression grammars. In: de Carvalho Junior, F.H., Barbosa, L.S. (eds.) SBLP 2012. LNCS, vol. 7554, pp. 72–86. Springer, Heidelberg (2012)
19. Reis, L.V.S., Bigonha, R.S., Di Iorio, V.O., Amorim, L.E.S.: The formalization and implementation of adaptable parsing expression grammars. Science of Computer Programming (to appear, 2014)
20. Schwerdfeger, A., Van Wyk, E.: Verifiable parse table composition for deterministic parsing. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 184–203. Springer, Heidelberg (2010)
21. Schwerdfeger, A.C., Van Wyk, E.R.: Verifiable composition of deterministic grammars. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 199–210. ACM, New York (2009)
22. Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., Nakatani, T.: Overview of the ibm java just-in-time compiler. IBM Syst. J. 39(1), 175–193 (2000)
23. Thompson, K.: Regular expression search algorithm. Commun. ACM 11(6), 419–422 (1968)
24. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Norwell (1985)
25. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (1997)
26. Warth, A., Piumarta, I.: OMeta: An Object-oriented Language for Pattern Matching. In: Proceedings of the 2007 Symposium on Dynamic Languages, DLS 2007, pp. 11–19. ACM, New York (2007)
27. Watt, D.A., Madsen, O.L.: Extended attribute grammars. Comput. J. 26(2), 142–153 (1983)
28. Wilson, G.V.: Extensible programming for the 21st century. Queue 2(9), 48–57 (2004)

A Security Types Preserving Compiler in Haskell

Cecilia Manzino[1] and Alberto Pardo[2]

[1] Universidad Nacional de Rosario, Rosario, Argentina
ceciliam@fceia.unr.edu.ar
[2] Universidad de la República, Montevideo, Uruguay
pardo@fing.edu.uy

Abstract. The analysis of information flow has become a popular technique for ensuring the confidentiality of data. An end-to-end confidentiality policy guarantees that private data cannot be inferred by the inspection of public data. A security property that ensures a kind of confidentiality is the noninterference property, which can be enforced by the use of security type systems where types correspond to security levels. In this paper we show the development of a compiler (written in Haskell) between a simple imperative language and semi-structured machine code, which preserves the property of noninterference. The compiler is based on the use of typed abstract syntax (implemented in terms of Haskell GADTs and type-level functions) to encode the security type system of both the source and target language. This makes it possible to use Haskell's type checker to verify two things: that programs in both languages satisfy the security property, and that the compiler is correct by construction (in the sense that it preserves noninterference).

1 Introduction

The confidentiality of the information manipulated by computing systems has became of significant importance with the increasing use of web applications. Even though these applications are widely used, there is little assurance that there is no leakage of confidential information to public output.

A technique that has been widely used in the last years for ensuring the confidentiality of information is the analysis of information flow [4]. This technique analyses information flows between inputs and outputs of systems. Flows can be explicit or implicit. A flow from a variable x to a variable y is considered explicit if the value of x is transferred directly to y. On the other hand, it is implicit when the flow from x to y is generated by the control structure of the program.

The security property we deal with in this paper is the so-called *noninterference property* [8]. This property guarantees that private data cannot be inferred by inspecting public channels of information. This implies that a variation of private data does not cause a variation of public data.

There are different approaches to ensure this security property. In this paper we follow a type-based approach which relies on the use of a security type system

F.M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 16–30, 2014.

[17]. In this setting, variables are augmented with labels indicating which kind of data they can store (for example, public or confidential). Modelling security properties in terms of types has the advantage that the property can be checked at compile-time (during type checking). The overhead of checking the property at run-time is thus partially reduced or even eliminated.

Although there is an important amount of work on type systems for noninterference on high-level languages (e.g. [20,17,1]), there is little work on type systems for noninterference on low-level languages. This is a consequence of the lack of structure of low-level languages, which make them difficult to reason about. Some of the existing works on security type systems for low-level languages use code annotations to simulate the block structure of high-level languages at the low-level [2,12]. Others are based on the use of a basic implicit structure present in low-level code [16].

The aim of this paper is to perform a simple, but not trivial excercise: showing that it is possible to write in a general purpose (functional) language like Haskell a compiler that preserves the property of noninterference. We do so by using Haskell plus some minor extensions, such as GADTs[1], type families and multi-parameter type classes, which open us the possibility to perform some kind of type-level programming. The compiler we present translates programs in a simple imperative high-level language (with loops and conditionals) into programs in a semi-structured low-level language. We define the notion of noninterference for each language by the specification of a security type system. Then we prove that the compiler preserves the noninterference property, i.e., programs that satisfy the security property in the source language are translated into programs that satisfy the security property in the target language. The novelty of the approach is that the proof of preservation is performed automatically by Haskell's type system. That way we are proving that the compiler is correct –in the sense that it preserves the property– by construction. This is a nonstandard application for Haskell which resembles a language with dependent types. We developed our implementation using the Glasgow Haskell Compiler (GHC).

The paper is organized as follows. In Section 2 we present the high-level language that serves as source language and show a security type system for it. We also describe how to encode the security type system as a GADT in Haskell. In section 3 we do the same with the low-level language that serves as target language. We also present a encoding of this language as a GADT in Haskell. Section 4 presents the compiler and the proof that it preserves security typing. Section 5 discusses related work and Section 6 concludes the paper.

2 Source Language

In this section, we introduce the high level language that is used as source language in the compilation. We start by describing its abstract syntax. Then we define its semantics and the type system used to enforce secure information

[1] Generalized Algebraic Data Types.

flow within programs. Finally, we show an implementation of both the syntax and the type system in Haskell.

2.1 Syntax

As source language we consider a simple imperative language whose expressions and sentences are defined by the following abstract syntax:

$$e ::= n \mid x \mid e_1 + e_2$$
$$S ::= x := e \mid \texttt{skip} \mid S_1;S_2 \mid \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } e \texttt{ do } S$$

where $e \in \mathbf{Exp}$ and $S \in \mathbf{Stm}$. Variables range over identifiers ($x \in \mathbf{Var}$) whereas n ranges over integer literals ($n \in \mathbf{Num}$).

2.2 Big-Step semantics

We present a semantics of the language which is completely standard [13]. In the semantics, the meaning of both expressions and statements is given relative to a state $s \in \mathbf{State} = \mathbf{Var} \to \mathbb{Z}$, a mapping from variables to integer values that contains the current value of each variable.

The semantics of expressions is given in terms of an evaluation function $\mathcal{E} : \mathbf{Exp} \to \mathbf{State} \to \mathbb{Z}$ defined by induction on the structure of expressions:

$$\begin{aligned} \mathcal{E}[\![n]\!]\, s &= \mathcal{N}[\![n]\!] \\ \mathcal{E}[\![x]\!]\, s &= s\, x \\ \mathcal{E}[\![e_1 + e_2]\!]\, s &= \mathcal{E}[\![e_1]\!]\, s + \mathcal{E}[\![e_2]\!]\, s \end{aligned}$$

where $\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$ is a function that associates an integer value to each integer literal.

For statements, we define a big-step semantics whose transition relation is written as $\langle S, s\rangle \Downarrow s'$, meaning that the evaluation of a statement S in an initial state s terminates with a final state s'. The definition of the transition relation is presented in Figure 1.

Notice that the language does not contain boolean expressions. In fact, the condition of an `if` as well as a `while` statement is given by an arithmetic expression. According to the semantics, the condition of an `if` statement is true when it evaluates to zero, and false otherwise. The same happens with the condition of a `while`.

2.3 Security Type System

We assume that each variable has associated a security level which states the confidentiality level of the values that it stores. For simplicity, in this paper we consider just two security levels, *low* and *high*, corresponding to *public* and *confidential* data, respectively, but the approach can be easily generalized to

$$\langle x := e,\, s\rangle \Downarrow s[x \mapsto \mathcal{E}[\![e]\!]\, s] \qquad \langle \texttt{skip},\, s\rangle \Downarrow s$$

$$\frac{\langle S_1,\, s\rangle \Downarrow s' \quad \langle S_2,\, s'\rangle \Downarrow s''}{\langle S_1;S_2,\, s\rangle \Downarrow s''}$$

$$\frac{\mathcal{E}[\![e]\!]\, s = 0 \quad \langle S_1,\, s\rangle \Downarrow s'}{\langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, s\rangle \Downarrow s'} \qquad \frac{\mathcal{E}[\![e]\!]\, s \neq 0 \quad \langle S_2,\, s\rangle \Downarrow s'}{\langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, s\rangle \Downarrow s'}$$

$$\frac{\mathcal{E}[\![e]\!]\, s = 0 \quad \langle S,\, s\rangle \Downarrow s' \quad \langle \texttt{while}\ e\ \texttt{do}\ S,\, s'\rangle \Downarrow s''}{\langle \texttt{while}\ e\ \texttt{do}\ S,\, s\rangle \Downarrow s''} \qquad \frac{\mathcal{E}[\![e]\!]\, s \neq 0}{\langle \texttt{while}\ e\ \texttt{do}\ S,\, s\rangle \Downarrow s}$$

Fig. 1. Big-step semantics of statements

a lattice of security levels ordered by their degree of confidentiality. As usual $low \leq high$.

We assume that the security level of each variable is maintained unchanged during program execution. In this sense, the language is said to be *flow insensitive*. We write x_L and x_H to mean a variable with *low* and *high* security level, respectively.

A program satisfies the noninterference property when the final value of the low variables is not influenced by a variation of the initial value of the high variables. This property can be formulated in terms of the semantics of the language. First we define that two states s and s' are said to be L-equivalent, written $s \cong_L s'$, when they coincide in the low variables, i.e., $s\, x_L = s'\, x_L$ for every low variable x_L. Hence, a program is noninterfering when, if executed in L-equivalent initial states and the execution terminates, it does so in L-equivalent final states:

$$\mathbf{NI}_S(S) \stackrel{\text{df}}{=} \forall s_i, s_i'.\, s_i \cong_L s_i' \wedge \langle S,\, s_i\rangle \Downarrow s_f \wedge \langle S,\, s_i'\rangle \Downarrow s_f' \implies s_f \cong_L s_f'$$

It is well-known that this property can be checked statically by the definition of an information-flow type system that enforces noninterference. We present such a type system for our language. It is based on similar type systems given in [20,17]. In the type system, security levels are used as types and are referred to as *security types*.

Expressions. For typing expressions we use a judgement of the form $\vdash e : st$, where $st \in \{low, high\}$. The rules for typing expressions is shown in Figure 2. Essentially, the security type of an expression is the maximum of the security types of its variables. We denote by $\texttt{max}\ st\ st'$ the maximum of two security types st and st'. Integer numerals are considered public data.

Statements. The goal of secure typing for statements is to prevent improper information flows at program execution. Information flow can appear in two forms: explicit or implicit. An *explicit flow* is observed when confidential data

EXPRESSIONS

$$\vdash n : low \qquad \vdash x_L : low \qquad \vdash x_H : high \qquad \frac{\vdash e : st \quad \vdash e' : st'}{\vdash e + e' : \texttt{max}\ st\ st'}$$

STATEMENTS

$$\frac{\vdash e : st}{[high] \vdash x_H := e}\ \text{ASSH} \qquad \frac{\vdash e : low}{[low] \vdash x_L := e}\ \text{ASSL}$$

$$[high] \vdash \texttt{skip}\ \ \text{SKIP} \qquad \frac{[pc_1] \vdash S_1 \quad [pc_2] \vdash S_2}{[\texttt{min}\ pc_1\ pc_2] \vdash S_1;S_2}\ \text{SEQ}$$

$$\frac{\vdash e : st \quad [pc_1] \vdash S_1 \quad [pc_2] \vdash S_2 \quad st \leq \texttt{min}\ pc_1\ pc_2}{[\texttt{min}\ pc_1\ pc_2] \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2}\ \text{IF}$$

$$\frac{\vdash e : st \quad [pc] \vdash S \quad st \leq pc}{[pc] \vdash \texttt{while}\ e\ \texttt{do}\ S}\ \text{WHILE}$$

Fig. 2. Security Type System of the Source Language

are copied to public variables. For example, the following assignment is not allowed because the value of a high variable is copied in a low variable.

$$y_L := x_H + 1$$

Implicit information flows arise from the control structure of the program. The following is an example of an insecure program where an implicit flow occurs:

$$\texttt{if}\ x_H\ \texttt{then}\ y_L := 1\ \texttt{else}\ \texttt{skip}$$

The reason for being insecure is because by observing the value of the low variable y_L we can infer whether the value of the high variable x_H is zero. The is a consequence of the assignment of a low variable in a branch of a conditional upon a high variable. Due to situations like this it is necessary to keep track of the security level of the program counter in order to know the security level of the context in which a sentence occurs.

The typing judgement for statements has the form $[pc] \vdash S$ and means that S is typable in the security context pc. The rules for typing statements are given in Figure 2. The type system turns out to be syntax-directed. Equivalent systems can be defined using a subsumption rule [20,17]. The reason for presenting a syntax-directed system is because it is the appropriate formulation to be considered for the implementation.

Rule ASSH states that assignments to high variables need to be performed in *high* contexts. In this type system a *high* context does not restrict typing, since by other rules (mainly those for conditional and loop) a code with *high* context can be part of any typable code. On the other hand, rule ASSL states that assignments to low variables can only be done in *low* contexts. Explicit

flows are prevented by this rule by the restriction to the expression to be low. The rules IF and WHILE impose a restriction between the security level of the condition and the branches (of the conditional) or the body (of the while). As a consequence, if the condition is high then the branches (of the conditional) or the body (of the while) must type in *high* contexts. This restriction together with that on assignments to low variables prevent implicit flows.

2.4 Implementation

We implement the security type system in Haskell by encoding the typing judgements as GADTs. A value of each GADT then represents a derivation of the encoded judgment. Using such an encoding the property (noninterference in our case) enforced by the type system of the object language is checked and maintained by the type system of the host language (Haskell in our case). This is a technique widely used nowadays [19].

Generalized Algebraic Data Types (GADTs) [15] are a generalization of the ordinary algebraic datatypes available in functional languages such as Haskell, ML or O'Caml. We explain the features that GADTs incorporate while showing the encoding of the type judgement corresponding to the arithmetic expressions of our language. To start, let us consider the abstract syntax definition for expressions. It can be represented in Haskell as the following datatype:

```
data Exp = IntLit Int | Var V | Add Exp Exp
```

where `V` is the type of variable names. The first ingredient that GADTs introduce is an alternative syntax for datatype declarations, where an explicit type signature is given for every data constructor. So we can define the `Exp` as:

```
data Exp where
  IntLit :: Int  → Exp
  Var    :: V    → Exp
  Add    :: Exp  → Exp  → Exp
```

The second feature that GADTs incorporate is even more important. GADTs remove the restriction present in parameterized algebraic datatypes by means of which the return type of every data constructor must be the same polymorphic instance of the type constructor being defined (i.e. the type constructor applied to exactly the same type variables as in the left-hand side of the datatype definition). In a GADT, on the contrary, the return type of the data constructors must still be an application of the same type constructor that is being defined, but the arguments can be arbitrary. This is the essential feature that makes it possible to encode type judgements as GADTs. It is not a minor point for these encoding the fact that the defined type systems are syntax-directed (i.e. type judgements reflect the structure of abstract syntax definitions).

We represent the type system for expressions as a GADT `Exp st`, similar to the one above for the abstract syntax but with the addition of a type parameter `st` that denotes the security type of the expression that a term encodes. The encoding is such that, the judgement $\vdash e : st$ in our formal type system

corresponds to the typing judgement `e :: Exp st` in Haskell. We represent the security types *low* and *high* in Haskell as empty types (i.e. datatypes with no constructors):

```
data Low
data High
```

The reason for using empty types is because we are only interested in computing with them at the type level. In fact, security types are only necessary to perform the static verification of the noninterference property on programs. They are not necessary at runtime.

The GADT for expressions is then the following:

```
data Exp st where
   IntLit :: Int     → Exp Low
   VarL   :: VL      → Exp Low
   VarH   :: VH      → Exp High
   Add    :: Exp st  → Exp st'  → Exp (Max st st')
```

where `VL` and `VH` are types for identifiers of low and high variables, respectively. The definition of disjoint sets for low and high variables (and consequently the definition of a constructor to each case) simplifies the implementation of the language, avoiding the necessity of supplying an environment with the security level of each variable. Notice that a separate treatment of each kind of variable had been already given in the definition of the formal type system.

In the encoding, the maximun of two security types is computed by means of the following type level function (called a *type family* [18] in Haskell's jargon):

```
type family Max st st' :: *
type instance Max Low x  = x
type instance Max High x = High
```

To model statements we define a GADT that is also parametrized by a security type, but in this case it represents the security level of the context in which a statement is executed.

```
data Stm pc where
   AssH  :: VH      → Exp st   → Stm High
   AssL  :: VL      → Exp Low  → Stm Low
   Skip  :: Stm High
   Seq   :: Stm pc  → Stm pc'  → Stm (Min pc pc')
   If    :: LEq st (Min pc pc')
         ⇒ Exp st  → Stm pc    → Stm pc'  → Stm (Min pc pc')
   While :: LEq st pc
         ⇒ Exp st  → Stm pc    → Stm pc
```

Each constructor corresponds to a rule of the type system shown in Figure 2. Now the typing judgement `stm :: Stm pc` corresponds to the judgement $[pc] \vdash S$ in the formal type system. It is worth noting that since `Stm pc` encodes the typing

rules, it is only possible to write terms that correspond to secure programs. Insecure programs will be rejected by the Haskell compiler because they correspond to ill-typed terms.

The minimum of two security types is computed by means of the following type level function:

```
type family Min st st' :: *
type instance Min Low x  = Low
type instance Min High x = x
```

The class `LEq` is defined for modelling the condition $pc \leq pc'$ at the type level, used in the typing rules for conditional and loop. This class has no operations.

```
class LEq a b
instance LEq Low  b
instance LEq High High
```

Having this class it is not necessary to provide a proof of the inequality between two types. If the condition holds for given two types, the selection of the appropriate instance will be chosen by Haskell's type system.

3 Target Language

The target language of the compiler is a simple stack machine in the style of the presented in [13]. In this section we describe its syntax and operational semantics and define a type system that guarantees noninterference.

3.1 Syntax

The code of the low-level language is given by the following abstract syntax:

$c ::=$	push n	pushes the value n on top of the stack
$\mid$	add	addition operation
$\mid$	fetch x	pushes the value of variable x onto the stack
$\mid$	store x	stores the top of the stack in variable x
$\mid$	noop	no operation
$\mid$	$c_1 ; c_2$	code sequence
$\mid$	branch (c_1, c_2)	conditional
$\mid$	loop (c_1, c_2)	looping

where $c \in \mathbf{Code}$, $x \in \mathbf{Var}$ and $n \in \mathbf{Num}$. Like the high-level language, this language also manipulates program variables that have associated a security level. As usual in this kind of low-level languages, values are placed in an evaluation stack in order to be used by operations (in this language, addition is the unique operation).

$$\langle \mathsf{push}\ n,\ \sigma,\ s\rangle \rhd (\mathcal{N}[\![n]\!] : \sigma,\ s) \qquad \langle \mathsf{add},\ z_1 : z_2 : \sigma,\ s\rangle \rhd ((z_1 + z_2) : \sigma,\ s)$$

$$\langle \mathsf{fetch}\ x,\ \sigma,\ s\rangle \rhd ((s\ x) : \sigma,\ s) \qquad \langle \mathsf{store}\ x,\ z : \sigma,\ s\rangle \rhd (\sigma, s[x \mapsto z])$$

$$\langle \mathsf{noop},\ \sigma,\ s\rangle \rhd (\sigma,\ s)$$

$$\frac{\langle c_1,\ \sigma,\ s\rangle \rhd (\sigma', s')}{\langle c_1\,;c_2,\ \sigma,\ s\rangle \rhd \langle c_2,\ \sigma',\ s'\rangle} \qquad \frac{\langle c_1,\ \sigma,\ s\rangle \rhd \langle c',\ \sigma',\ s'\rangle}{\langle c_1\,;c_2,\ \sigma,\ s\rangle \rhd \langle c'\,;c_2,\ \sigma',\ s'\rangle}$$

$$\frac{z = 0}{\langle \mathsf{branch}\ (c_1, c_2),\ z : \sigma,\ s\rangle \rhd \langle c_1,\ \sigma,\ s\rangle} \qquad \frac{z \neq 0}{\langle \mathsf{branch}\ (c_1, c_2),\ z : \sigma,\ s\rangle \rhd \langle c_2,\ \sigma,\ s\rangle}$$

$$\langle \mathsf{loop}\ (c_1, c_2),\ \sigma,\ s\rangle \rhd \langle c_1\,;\mathsf{branch}\ (c_2\,;\mathsf{loop}\ (c_1, c_2), \mathsf{noop}),\ \sigma,\ s\rangle$$

Fig. 3. Operational Semantics of the Target Language

3.2 Operational Semantics

A code c is executed on an abstract machine with configurations of the form $\langle c, e, s\rangle$ or (e, s), where σ is an evaluation stack and $s \in \mathit{State}$ is a state that associates values to variables. The operational semantics is given by a transition relation between configurations that specifies an individual execution step. The transition relation is of the form $\langle c,\ \sigma,\ s\rangle \rhd \gamma$, where γ may be either a new configuration $\langle c',\ \sigma',\ s'\rangle$, expressing that remaining execution steps still need to be performed, or a final configuration (σ', s'), expressing that the execution of c terminates in one step. As usual, we write $\langle c,\ \sigma,\ s\rangle \rhd^* \gamma$ to indicate that there is a finite number of steps in the execution from $\langle c,\ \sigma,\ s\rangle$ to γ. The operational semantics of the language is shown in Figure 3.

We can define a *meaning relation* $\langle c, s\rangle \downarrow s'$ *iff* $\langle c,\ \epsilon,\ s\rangle \rhd^* (\sigma', s')$ which states that a given code c, and states s and s' are in the relation whenever the execution of c starting in s and the empty stack ϵ terminates with state s'. It can be proved that this is in fact a partial function as our semantics is deterministic. Based on this relation we can define what does it mean for a low-level program to be noninterfering:

$$\mathbf{NI}_T(c) \stackrel{\text{df}}{=} \forall s_i, s_i'.\ s_i \cong_L s_i' \ \wedge\ \langle c, s_i\rangle \downarrow s_f \ \wedge\ \langle c, s_i'\rangle \downarrow s_f' \implies s_f \cong_L s_f'$$

3.3 Security Type System

The security type system is defined in terms of a transition relation that relates the security level of the program counter with the state of a stack of security types before and after the execution of a program code. The typing judgement is then of the form $ls \vdash c : pc \rightsquigarrow ls'$, where pc is the security level of the program counter, and ls and ls' are stacks of security types. This judgement states that a program c is typable when it is executed in the security environment given by the stack ls, the program counter has security level pc and it ends with stack ls'. The type system is shown in Figure 4.

$$\textsc{push}\quad ls \vdash \mathsf{push}\ n : high \rightsquigarrow Low :: ls$$

$$\textsc{add}\quad st_1 :: st_2 :: ls \vdash \mathsf{add} : high \rightsquigarrow \mathtt{max}\ st1\ st2 :: ls$$

$$\textsc{fetchL}\quad ls \vdash \mathsf{fetch}\ x_L : high \rightsquigarrow Low :: ls$$

$$\textsc{fetchH}\quad ls \vdash \mathsf{fetch}\ x_H : high \rightsquigarrow High :: ls$$

$$\textsc{storeL}\quad Low :: ls \vdash \mathsf{store}\ x_L : Low \rightsquigarrow ls$$

$$\textsc{storeH}\quad st :: ls \vdash \mathsf{store}\ x_H : high \rightsquigarrow ls$$

$$\textsc{noop}\quad ls \vdash \mathsf{noop} : high \rightsquigarrow ls$$

$$\textsc{cseq}\ \frac{ls \vdash c_1 : pc_1 \rightsquigarrow ls' \qquad ls' \vdash c_2 : pc_2 \rightsquigarrow ls''}{ls \vdash c_1\,;c_2 : \mathtt{min}\ pc_1\ pc_2 \rightsquigarrow ls''}$$

$$\textsc{branch}\ \frac{ls \vdash c_1 : pc_1 \rightsquigarrow ls \qquad ls \vdash c_2 : pc_2 \rightsquigarrow ls \qquad st \leq \mathtt{min}\ pc_1\ pc_2}{st :: ls \vdash \mathsf{branch}\ (c_1, c_2) : \mathtt{min}\ pc_1\ pc_2 \rightsquigarrow ls}$$

$$\textsc{loop}\ \frac{ls \vdash c_1 : pc_1 \rightsquigarrow st :: ls' \qquad ls' \vdash c_2 : pc_2 \rightsquigarrow ls' \qquad st \leq pc_2}{ls \vdash \mathsf{loop}\ (c_1, c_2) : \mathtt{min}\ pc_1\ pc_2 \rightsquigarrow ls'}$$

Fig. 4. Security Type System for the Target Language

Like for the high-level language, this type system was designed in order to prevent explicit and implicit illegal flows. Rule STOREL, for example, prevents explicit flows by requiring that the value to be stored in a variable low has also security level *low*, while the requirement on the context (which must be *low*) prevents implicit flows. Rules BRANCH and LOOP also take care of implicit flows. Rule BRANCH, for example, requires that the security level of the program counters of the branches must be at least the security level of the value at the top of the stack (which is the value used to choose the branch to continue). Something similar happens with the loop construct in rule LOOP.

We note that this type system rejects some secure programs. For example, the following program is not accepted:

$$\mathsf{push}\ 1; \mathsf{branch}\ (\mathsf{push}\ 0, \mathsf{noop})$$

because of the restriction of rule BRANCH, which states that the branches of a conditional cannot change the stack of security types. In this case, the branch push 0, adds an element to the stack. If we decided to remove such a restriction, we should be careful that a code like the following one, clearly insecure, is rejected by the type system:

$$\mathsf{fetch}\ x_H; \mathsf{branch}\ (\mathsf{push}\ 1, \mathsf{push}\ 2); \mathsf{store}\ x_L$$

However, without the restriction of rule BRANCH this is a situation not easy to detect because the instruction store x_L occurs outside the conditional and

depends on the code that comes before it. This problem is due to the way in which the storage of a value in a variable is performed in this language. In fact, at least two actions are required, rather than one: one for allocating the value (to be stored in the variable) in the stack, and another for moving that value to the variable.

A similar situation happens with rule LOOP. It rejects any loop $\mathsf{loop}\ (c_1, c_2)$ whose body c_2 changes the stack of security types.

In Section 4 we show that the programs which, even secure, are rejected by these restrictions of the type system are not the ones generated by compilation.

3.4 Implementation

We use type-level lists to represent the stacks of security types. Such lists can be defined by introducing the following empty types:

```
data Empty
data st :#: l
```

The type `Empty` represents the empty stack whereas a type `st :#: l` represents a stack with top element `st` and tail stack `l`.

The low-level language is encoded by a GADT that is parameterized by the security level of the context and the stacks of types before and after the execution of the code.

```
data Code ls pc ls' where
   Push   :: Int  →  Code ls High (Low :#: ls)
   AddOp  :: Code (st1 :#: st2 :#: ls) High (Max st2 st1 :#: ls)
   FetchL :: VL   →  Code ls High (Low :#: ls)
   FetchH :: VH   →  Code ls High (High :#: ls)
   StoreL :: VL   →  Code (Low :#: ls) Low ls
   StoreH :: VH   →  Code (pc :#: ls) High ls
   Noop   :: Code ls High ls
   CSeq   :: Code ls pc1 ls'  →  Code ls' pc2 ls''
          →  Code ls (Min pc1 pc2) ls''
   Branch :: LEq  pc (Min pc1 pc2)
          ⇒ Code ls pc1 ls  →  Code ls pc2 ls
           →  Code (pc :#: ls) (Min pc1 pc2) ls
   Loop   :: LEq st pc2
          ⇒ Code ls pc1 (st :#: ls')  →  CodeS ls' pc2 ls'
          →  Code ls (Min pc1 pc2) ls'
```

The typing judgement `c :: Code ls pc ls'` corresponds to the judgement $ls \vdash c : pc \rightsquigarrow ls'$ in the formal type system.

4 Compiler

The compiler is a function that converts terms of the source language into terms of the target language. Since the terms of our source language are of two syntax

Expressions

$$\boldsymbol{C}_e[n] = \mathsf{push}\ n$$
$$\boldsymbol{C}_e[x] = \mathsf{fetch}\ x$$
$$\boldsymbol{C}_e[e_1 + e_2] = \boldsymbol{C}_e[e_1]\ ;\ \boldsymbol{C}_e[e_2]\ ;\ \mathsf{add}$$

Sentences

$$\boldsymbol{C}_S[x := e] = \boldsymbol{C}_e[e]\,;\mathsf{store}\ x$$
$$\boldsymbol{C}_S[\mathtt{skip}] = \mathsf{noop}$$
$$\boldsymbol{C}_S[S_1;S_2] = \boldsymbol{C}_S[S_1]\,;\,\boldsymbol{C}_S[S_2]$$
$$\boldsymbol{C}_S[\mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2] = \boldsymbol{C}_e[e]\,;\mathit{branch}(\boldsymbol{C}_S[S_1],\,\boldsymbol{C}_S[S_2])$$
$$\boldsymbol{C}_S[\mathtt{while}\ e\ \mathtt{do}\ S] = \mathit{loop}(\boldsymbol{C}_e[e],\,\boldsymbol{C}_S[S])$$

Fig. 5. Compilation functions

categories, we have to define two compilation functions, one for expressions ($\boldsymbol{C}_e$: **Exp** $\rightarrow$ **Code**) and the other for commands ($\boldsymbol{C}_S$: **Stm** $\rightarrow$ **Code**). Figure 5 shows the definition of both functions.

It is not difficult to prove that this compiler is correct with respect to the semantics of the source and target languages.

Theorem 1 (COMPILER CORRECTNESS). *For any expression e, statement S of the source language, and state s it holds that:*

i) $\langle \boldsymbol{C}_e[e],\ \epsilon,\ s\rangle \rhd^* (\mathcal{E}[\![e]\!]s, s)$

ii) if $\langle S,\ s\rangle \Downarrow s'$ *then* $\langle \boldsymbol{C}_S[S],\ \epsilon,\ s\rangle \rhd^* (\epsilon, s')$

However, in this paper we are especially interested in another property of the compiler, namely the preservation of the noninterference by compilation. This means that, if we start with a noninterfering program in the source language, then the compiler returns a noninterfering program in the target language. We can express this property semantically.

Theorem 2 (PRESERVATION OF NONINTERFERENCE). *For any expression e and statement S,*

i) $\mathbf{NI}_T(\boldsymbol{C}_e[e])$

ii) if $\mathbf{NI}_S(S)$ *then* $\mathbf{NI}_T(\boldsymbol{C}_S[S])$

However, our interest is to establish this property in terms of the type systems.

Theorem 3 (TYPE-BASED PRESERVATION OF NONINTERFERENCE). *For any expression e and statement S,*

i) If $\vdash e : st$ *then* $ls \vdash \boldsymbol{C}_e[e] : high \rightsquigarrow st :: ls$

ii) If $[pc] \vdash S$ *then* $ls \vdash \boldsymbol{C}_S[S] : pc \rightsquigarrow ls$

Proof. By induction on the structure of expressions and statements.

4.1 Implementation

Both $\boldsymbol{C}_e$ and $\boldsymbol{C}_S$ can be easily implemented in Haskell.

```
compE :: Exp st → Code ls High (st :#: ls)
compE (IntLit n)  =  Push n
compE (VarL x)    =  FetchL x
compE (VarH x)    =  FetchH x
compE (Add e1 e2) =  CSeq (CSeq (compE e1) (compE e2)) AddOp

compS :: Stm pc → Code ls pc ls
compS (AssL x e)   =  CSeq (compE e) (StoreL x)
compS (AssH x e)   =  CSeq (compE e) (StoreH x)
compS Skip         =  Noop
compS (Seq s1 s2)  =  CSeq (compS s1) (compS s2)
compS (If e s1 s2) =  CSeq (compE e) (Branch (compC s1) (compC s2))
comps (While e s)  =  Loop (compE e) (compS s)
```

However, we should not forget that as GADTs we have represented not simply the abstract syntax of the languages but actually their secure type systems. Therefore, these translation functions turn out to be more than compilation functions. They are actually the Haskell representation of the proof terms of Theorem 3! In fact, observe that the type of these functions is exactly the Haskell encoding of the properties *i)* and *ii)* in the Theorem. In other words, when writing these functions we are actually proving this Theorem and the verification that we are doing so is done by Haskell type system. As we mentioned above, both *i)* and *ii)* are proved by structural induction. The different equations of the translation functions encode the cases of those inductive proofs.

This is a common situation in languages with dependent types, like Agda [14], but was not so in languages like Haskell. However, with the increasing incorporation of new features to Haskell (and GHC) this sort of type level programming applications are becoming more frequent and feasible.

5 Related Work

There has been a lot of work on information flow analysis, pioneered by Bell and LaPadula [4], and continued with the work of Denning [6]. Noninterference was introduced by Goguen and Meseguer [8]. One of the approaches to ensure this security property has been based on the use of type system [6,17]. This is the approach we followed in this paper. Most of the works on type systems for noninterference concentrated on high-level languages (e.g. [20,17,1]), but there are also some works that studied security type systems for low-level languages (e.g. [2,12,16]).

In this paper we used a limited form of dependent type programming available in Haskell and in particular in the GHC compiler by the use of some extensions. The frontiers of dependent-type programming in Haskell is nowadays a subject of discussion and experimentation (see e.g. [7,11]).

There are some works on the use of dependent types for developing type-preserving compilers. Chlipala [5], for example, developed a certified compiler from the simply-typed lambda calculus to an assembly language using the proof assistant Coq. He uses dependent types in the representation of the target language of its compiler to ensure that only terms with object language typing derivations are representable.

Guillemette and Monnier [9] wrote a type-preserving compiler for System F in Haskell. Their compiler is composed by phases so that Haskell's type checker can mechanically verify the typing preservation of each phase.

An example of the use of GADTs for ensuring static properties of programs is presented by Tim Sheard [19], who wrote a simple While language that satisfied two semantic properties: scoping and type safety. He presented this example in the language Omega.

These works are similar to ours in the sense that they prove that a compiler preserves the types of object programs, or that a language satisfies some static property, but they are not concerned with proving the preservation of a security property of programs.

There are many works on certified compilers. One is that by Leroy [10] who wrote a certified compiler for a subset of C in the proof assistant Coq. Another is the work by Barthe, Naumann and Rezk [3] who wrote a compiler for Java that preserves information flow typing, such that any typable program is compiled into a program that will be accepted by a bytecode verifier that enforces noninterference.

6 Conclusion

We presented a compiler written in Haskell that preserves the property of noninterference. The compiler takes code in an imperative high-level language and returns code in a low-level language that runs in a stack-based abstract machine. For each of the languages we defined the property of noninterference by means of a security type system. Those type systems were then represented in Haskell in terms of GADTs combined with type families for computing with security types at the type level and a multi-parameter type class for comparing security types. This encoding guarantees that we can only write noninterfering programs (i.e. valid terms of the corresponding GADTs).

Using this approach the type of the compiler corresponds exactly to the formulation of the property that noninterference is preserved by compilation. The definition of the compiler function itself then corresponds to the proof term that proves that property. The rest of the work (i.e. the verification that the function is indeed a proof of that property) is is done by Haskell's type system.

Although the target language of the compiler is simple and semi-structured, we think that a compiler to a more realistic low-level language (for example, with goto) can be constructed applying the same ideas. We are currently doing some experiments in this line using Agda [14], but we do not discard to try with Haskell as well. The decision of using Agda for developing a more realistic compiler is because the complexity of the required program properties increase and then it becomes difficult to express them in Haskell.

References

1. Banerjee, A., Naumann, D.A.: Stack-based access control and secure information flow. J. Funct. Program. 15(2), 131–177 (2005)
2. Barthe, G., Rezk, T., Basu, A.: Security types preserving compilation. Comput. Lang. Syst. Struct. 33(2), 35–59 (2007)
3. Barthe, G., Rezk, T., Naumann, D.A.: Deriving an information flow checker and certifying compiler for java. In: IEEE Symposium on Security and Privacy, pp. 230–242. IEEE Computer Society (2006)
4. Bell, D.E., LaPadula, L.J.: Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, The MITRE Corp. (1975)
5. Chlipala, A.: A certified type-preserving compiler from lambda calculus to assembly language. SIGPLAN Not. 42(6), 54–65 (2007)
6. Denning, D.E.: A lattice model of secure information flow. Commun. ACM 19(5), 236–243 (1976)
7. Eisenberg, R.A., Weirich, S.: Dependently typed programming with singletons. In: Proceedings of the 2012 Haskell Symposium, Haskell 2012, pp. 117–130. ACM, New York (2012)
8. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Symposium on Security and Privacy, pp. 11–20. IEEE Computer Society Press (1982)
9. Guillemette, L.-J., Monnier, S.: A type-preserving compiler in haskell. SIGPLAN Not. 43(9), 75–86 (2008)
10. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. 43(4), 363–446 (2009)
11. Lindley, S., McBride, C.: Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming. In: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Haskell 2013, pp. 81–92. ACM, New York (2013)
12. Medel, R., Compagnoni, A.B., Bonelli, E.: A typed assembly language for noninterference. In: Coppo, M., Lodi, E., Pinna, G.M. (eds.) ICTCS 2005. LNCS, vol. 3701, pp. 360–374. Springer, Heidelberg (2005)
13. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction. John Wiley & Sons, Inc., New York (1992)
14. Norell, U.: Dependently typed programming in agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009)
15. Jones, S.P., Vytiniotis, D., Weirich, S., Washburn, G.: Simple Unification-based Type Inference for GADTs. In: 11th International Conference on Functional Programming, pp. 50–61. ACM (2006)
16. Saabas, A., Uustalu, T.: Compositional type systems for stack-based low-level languages. In: Proceedings of the 12th Computing: The Australasian Theroy Symposium, CATS 2006, vol. 51, pp. 27–39. Australian Computer Society, Inc., Darlinghurst (2006)
17. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communications 21(1), 5–19 (2003)
18. Schrijvers, T., Jones, S.P., Chakravarty, M., Sulzmann, M.: Type checking with open type functions. SIGPLAN Not. 43(9), 51–62 (2008)
19. Sheard, T.: Languages of the future. SIGPLAN Not. 39(12), 119–132 (2004)
20. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. 4(2-3), 167–187 (1996)

Avoiding Code Pitfalls in Aspect-Oriented Programming

Péricles Alves[1], Eduardo Figueiredo[1], and Fabiano Ferrari[2]

[1] Software Engineering Laboratory, Federal University of Minas Gerais (UFMG), Brazil
[2] Computing Department, Federal University of São Carlos (UFSCar), Brazil
{periclesrafael,figueiredo}@dcc.ufmg.br, fabiano@dc.ufscar.br

Abstract. Aspect-Oriented Programming (AOP) is a maturing technique that requires a good comprehension of which types of mistakes programmers make during the development of applications. Unfortunately, the lack of such knowledge seems to represent one of the reasons for the cautious adoption of AOP in real software development projects. Based on a series of experiments, this paper reports a catalogue of code pitfalls that are likely to lead programmers to make mistakes in AOP. Each experiment required the aspectization (i.e. refactoring) of a crosscutting concern in one object-oriented application. Six rounds of the experiment provided us with the data of 80 aspect-oriented (AO) implementations where three crosscutting concerns were aspectized in three applications. We developed a prototype tool to warn programmers of the code pitfalls during refactoring activities.

Keywords: AOP, mistakes, code pitfalls, empirical study.

1 Introduction

Aspect-Oriented Programming (AOP) [19] is a software development technique that aims to improve software modularity through the separation of crosscutting concerns into modular units called aspects. A concern is any consideration that can impact on the design and maintenance of program modules [28]. It is well known that novice programmers need special guidance while learning how to program with a new language [30, 31]. Therefore, to be widely adopted in practice, we need a good comprehension of the kinds of mistakes made by AOP programmers when learning this development technique and the situations that lead to these mistakes.

A mistake is "a human action that produces an incorrect result" [17]. In our study, a mistake occurs when a developer performs an inconsistent code refactoring, which may result in a fault into the application. A fault, on the other hand, consists in an incorrect step, process, or data definition in a computer program [17]. In other words, a fault occurs when there is a difference between the actually implemented software product and the product that is assumed to be correct. A mistake may lead to one or more faults being inserted into the software, although it not necessarily does so.

Previous research [3, 5, 9, 11] investigated mistakes that are likely to be made by AOP programmers either to build systems from scratch or to refactor existing ones. Other studies identified AO code smells [21, 24, 26]. However, these studies often

F.M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 31–46, 2014.

consider complex systems developed by experienced programmers. In such scenarios, it is difficult to reveal the factors that hinder the learning of basic AOP concepts, such as pointcut and advice, by novice programmers. This situation gives us a lack of understanding of the scenarios that could lead novice programmers to make mistakes while refactoring existing code to AOP. In turn, it may represent one of the reasons for the cautious adoption of AOP in real software development projects [32].

Towards addressing this problem, we identified a preliminary set of recurring mistakes made by novice AOP programmers in our previous work [4]. We derived these mistakes by running three rounds of an experiment in which 38 novice AOP programmers were asked to refactor out to aspects two crosscutting concerns from two small Java applications. The mistakes observed in our previous study are often related to fault types documented in previous research [3, 5, 9, 11]. We also documented new kinds of programming mistakes, such as Incomplete Refactoring, which does not necessary lead to faults.

This paper extends our previous study [4] and builds up on top of its results. First, this paper reports on other three rounds of the same experiment (resulting in six rounds in total) with a different application (called Telecom) and 42 additional participants. Therefore, this paper relies on data of 80 refactored code samples of three small-sized applications (Section 2). In this follow up study, our goal is not only to further investigate and confirm our previous findings, but also to find out additional categories of common mistakes made by AOP programmers. Second, this paper also presents a novel catalogue of situations that appear to lead programmers to make the identified categories of recurring mistakes (Section 3). Such situations, named Aspectization Code Pitfalls, were observed by inspecting the original source code focusing on code fragments that were incorrectly or inconsistently refactored to AOP. To support the automatic detection of the documented Aspectization Code Pitfalls, we propose a prototype tool called ConcernReCS (Section 4). Section 5 summarizes related work and Section 6 concludes this paper with directions for future work.

2 Empirical Study

This section presents the configurations of the experimental study we conducted. Section 2.1 presents its research questions and Section 2.2 briefly describes the study participants. Section 2.3 introduces the target applications and characteristics of the refactored crosscutting concerns. Section 2.4 explains the experimental tasks.

2.1 Research Questions

This study aims at investigating the types of mistakes made by students and junior professionals when using AOP. Our goal in this study is to uncover and document code pitfalls that lead programmers to make these mistakes. Based on this goal, we formulate the research questions below. To answer RQ1, we first identify and classify the recurring categories of mistakes made by programmers learning AOP. Then, we document error-prone situations as a catalogue of code pitfalls to address RQ2.

RQ1. What kinds of mistakes do novice AOP programmers often make while refactoring crosscutting concerns?

RQ2. What are the code pitfalls in the source code that lead to these mistakes?

2.2 Background of Subjects

Participants of all six rounds of this study were organized in groups of one or two members, called subjects of the experiment. In three rounds, participants worked in pairs in order to investigate possible impact of pair programming on the mistakes [4].

Table 1 summarizes the number of subjects (i.e., groups of participants) and how they were organized in each study round. Each subject took part of only one round of the experiment. Note that in the odd rounds (1st, 3rd, and 5th), the numbers of participants are twice the number of subjects since each subject includes two participants. That is, the total number of participants is 108 divided into 80 subjects. Table 1 also shows the application each subject worked with. That is, subjects of the first two rounds refactored a concern of an ATM application, of the following two rounds refactored a Chess game, and of the last two rounds refactored a Telecom system. We describe the software systems and the refactored concerns in Section 2.3.

Table 1. Number of subjects in each round

System	Round	# of Subjects	Grouping
ATM	First	11	Pairs
	Second	17	Individually
Chess	Third	15	Pairs
	Fourth	15	Individually
Telecom	Fifth	3	Pairs
	Sixth	19	Individually
Total number of subjects		**80**	

The participants filled in a questionnaire with their background information. The answers regard participants' level of knowledge in Object-Oriented Programming (OOP) and their work experience in software development. This questionnaire aims to characterize the background of the study participants. In general, more than 80% of participants claimed to have some experience in OOP. Since most participants are undergraduate or graduate students, about half of them have never worked in a software development company. Due to space constraints, we provide detailed information of the participants' backgrounds in the project website [1].

2.3 Target Applications and Crosscutting Concerns

Three small Java applications were chosen to be used in this study: ATM, Chess, and Telecom. Each participant were asked to refactor one crosscutting concern of each application using the AspectJ programming language [18]. Table 2 summarizes some

size measurement for the target applications and their respective crosscutting concerns. The Number of Classes (NC) and Lines of Code (LOC) metrics indicate the size of each application in terms of classes/interfaces and lines of code, respectively. Additionally, Concern Diffusion over Classes (CDC) [12, 14] and Lines of Concern Code (LOCC) [12, 14] measure the concern size using the same units. All concerns and applications share similar complexity and were carefully chosen to allow subjects to finish the experimental task within 90 minutes. Furthermore, we chose simple applications with distinct characteristics and from different domains to allow us assessing the complexity behind heterogeneous AOP constructs, such as inter-type declaration, pointcut, and advice [18].

Table 2. Size metrics of the target applications and concerns

Application/ concern	Application size		Concern size	
	NC	LOC	CDC	LOC
ATM / Logging	12	606	4	19
Chess / ErrorMessages	13	1011	8	36
Telecom / Timing	8	213	4	32

The ATM application simulates three basic functionalities of an ordinary cash machine: show balance, deposit, and withdraw. Subjects of the first two rounds had to refactor out to aspects the Logging concern in this application. This concern is responsible for recording all operations performed by an ATM user. It is implemented by 19 LOC diffused over 4 application modules. The Chess application implements a chess game with a graphical user interface. Subjects of the third and fourth rounds refactored the ErrorMessages concern of this application. This concern is responsible for displaying messages when a player tries to break the chess rules. This concern is implemented by 36 LOC spread over 8 classes. Finally, Telecom is a connection management system for phone calls implemented in Java and AspectJ [2]. It simulates a call between two or more customers. We only used the Java implementation and asked each subject of the last two rounds to refactor out to aspects the Timing concern. This concern is responsible for measuring the elapsed time of a given call. It is implemented in 30 LOC spread over 4 modules.

2.4 Experimental Tasks

Before performing the experimental tasks, all participants attended a 2-hour training session about AOP and AspectJ. After the training session, each group of participants received the source code of a small Java application and a textual description of the crosscutting concern. Using the Eclipse IDE (version 3.7 Indigo) with the AJDT plug-in (version 2.1.3) properly installed, subjects were asked to refactor the crosscutting concern from the given application using the AspectJ language constructs. Note that the subjects were responsible for both correctly identifying the concern code in the application as well as choosing the proper AspectJ constructs to refactor the given crosscutting concern. No instruction was given in this sense because we consider

these tasks part of the AOP programmer regular duties. After subjects concluded their tasks, we performed our analysis in two steps. First, we investigated which were the categories of AOP-specific mistakes the programmers frequently made (Section 3). Then, we identified situations in the OO original code, named code pitfalls (Section 4), that may lead programmers to make mistakes.

3 Recurring Mistakes in AOP

This section summarizes the results of our study and classifies the mistakes recurrently made by subjects during the experimental tasks. This classification is based on our previous work [4]. Mistakes were identified by manual code inspection.

3.1 Classification of Recurring Mistakes

Incorrect Implementation Logic. It occurs when a piece of concern code is aspectized through pointcut and advice mechanisms. However, the refactored code behaves differently from the original code. For instance, by selecting an option in the main menu of ATM, a logging report of that operation can be displayed on screen. However, novice programmers often make this kind of mistake because it involves an intricate concern code nested in switch and while constructs.

Incorrect Advice Type. In this mistake category, the programmer uses an incorrect type of advice to refactor a piece of concern code. For instance, in the OO version of the ATM application, the message that records information about the `deposit` operation is logged at the end of the `credit()` method of the `BankDatabase` class. This operation should therefore be refactored as an after advice. However, some subjects of this study wrongly use a before advice to refactor this code.

Compilation Error or Warning. It refers to either a syntactic fault or a potential fault signaled by the AspectJ compiler through an error or warning message[1]. For instance, in the OO version of the ATM application, the logging message that records the user authentication is stored when the `authenticateUser()` method of the `BankDatabase` class is executed. Although this method has two integer parameters, some subjects have not specified the method parameters neither used a wildcard when writing the pointcut to select this join point. Consequently, a warning message notifies the programmer that no join point matched this specific pointcut.

Duplicated Crosscutting Code. In this mistake, part of the code that implements a crosscutting concern appears replicated in both aspects and the base Java code. For instance, the `getErrorMsg()` method of the `ChessPiece` class implements part of ErrorMessages and should be refactored to aspects in the Chess application. However,

[1] Note that, sometimes syntactic problems in AspectJ code are not promptly signaled by the compiler. Programmers can only notice such problems when they perform a full weaving. Since programmers usually work with tied schedule, they do not always double-check their code to remove these types of faults.

this piece of code is sometimes not only implemented into aspects but also left in the original class (i.e., `ChessPiece`). Therefore, this mistake results in duplicated code.

Incomplete Refactoring. This mistake occurs when developers fail to refactor part of the crosscutting concern. As a result, part of concern still remains in the base code in the AOP solution. This mistake was made, for instance, in the aspectization of the `totalConnectionTime` attribute in the `Costumer` class that implements part of Timing in the Telecom application. When refactoring this concern, some subjects have not transferred this attribute to an aspect.

Excessive Refactoring. This kind of mistake is made when part of the base code that does not belong to a concern is refactored to aspects. As an example, when refactoring the Timing concern in Telecom, developers should move the `timer.start()` call from the `complete()` method in `Connection` to an advice. However, some subjects also refactored the previous statement: `System.out.println("connection completed")`. Although this statement also appears in the `complete()` method, it should not be refactored since it does not belong to the Timing concern.

3.2 Experimental Results

Our analysis is based on 80 AOP implementations of the three target applications (Section 2.3). AOP-related mistakes in these implementations were analyzed and classified according to the categories discussed in Section 3.1. Figure 1 presents the overall percentage of subjects that made mistakes within each category taking each application into consideration. Subjects were related to a given category if they made at least one mistake within that category regardless of the absolute number of mistakes. We made this decision because it would be hard to quantify how many times a subject makes some kinds of mistakes, such as Incomplete Refactoring, due to the higher granularity of them. Therefore, counting individual instances of a mistake could give us the wrong impression that a mistake (e.g., Incomplete Refactoring) is less frequent than others (e.g., Incorrect Advice Type).

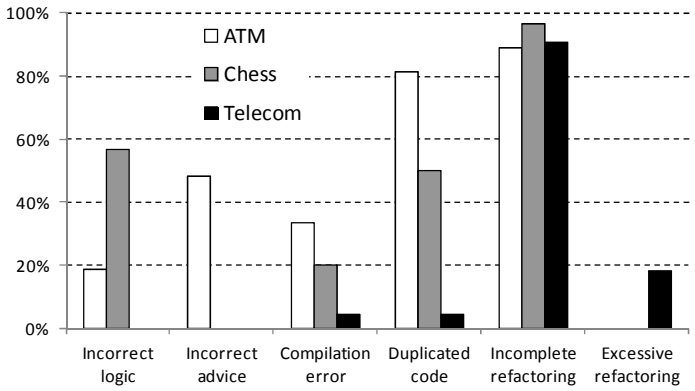


Fig. 1. Subjects that made mistakes in each category

Figure 1 shows that Incomplete Refactoring is the most common mistake in all applications. This mistake occurred with fairly the same rate (about 95%) in all applications. Unlike Incomplete Refactoring, some mistakes are more common in one application than in the others. For instance, Duplicated Crosscutting Code and Incorrect Advice Type are common in the ATM application. Incorrect Implementation Logic, on the other hand, is more frequent in the Chess application. This result is mainly due to (i) the crosscutting nature of the aspectized concern and (ii) properties of the target application. For instance, no subject has made the Incorrect Advice Type mistake in the Chess and Telecom applications. After inspecting the concern code, we observed that all advices used to aspectize ErrorMessages (Chess) and Timing (Telecom) should be after advices. Therefore, due to the homogeneity of the concern code, it is unlike that someone would use a different type of advice and make this kind of mistake in these two concern instances.

Incomplete Refactoring is often made in all sorts of systems due to at least two major reasons: inability to identify the concern code and inability to separate it. In the former case, programmers fail to assign code elements to the concern that should actually be later refactored. Nunes *et al*. [25] highlighted that the inability to identify the concern code is one of the most problematic steps for concern refactoring. We empirically confirm their results. Regarding the inability to separate concern code, we found out that programmers avoid refactoring pieces of code if such code is very tangled; in such cases, programmers end up leaving it mixed up with the base code.

The Excessive Refactoring mistake only happened in the Telecom application (see Figure 1). We observed that this kind of mistake seems to be less often in software aspectization. In fact, the particular way Timing is implemented in Telecom might have led programmers to make Excessive Refactoring. For instance, Figure 2 shows the partial OO implementation of the `BasicSimulation` class in the Telecom application. The grey shadow indicates a line of code realizing the Timing concern. Programmers are expected to use pointcut-advice to aspectize this part of concern. However, some programmers moved this method from the `BasicSimulation` class to an aspect and introduced it back by means of inter-type declaration; that is, they aspectized the whole method instead of just a single line of code. This mistake occurred mainly because the `report()` method has a solo line of code. Therefore, OO programmers feel uncomfortable with an empty method in the application and prefer moving it to an aspect.

```
public class BasicSimulation extends AbstractSimulation {
...
  protected void report(Customer c) {
     System.out.println(c+"spent"+c.getTotalConnectTime());
  }
}
```

Fig. 2. Timing implementation in the BasicSimulation class

4 Aspectization Code Pitfalls

After collecting the mistakes made by the subjects of the experiment described in Section 3, we manually inspect situations that could have led them to recurrently make these mistakes. We name these situations *code pitfalls* and analyzed them in two phases. Firstly, we performed a manual inspection in the OO source code of the applications in order to identify code pitfalls for AO refactoring. Secondly, we interviewed some of the experiment participants with the purpose to verify whether such code pitfalls may have impacted on the mistakes they made. The result of this investigation is a preliminary catalogue of code pitfalls described in this section. We organized these code pitfalls into two major categories: Dedicated Implementation Elements (Section 5.1) and Non-Dedicated Implementation Elements (Section 5.2).

4.1 Dedicated Implementation Elements

This section discusses two code pitfalls related to dedicated implementation elements, namely *Primitive Constant* and *Attribute of a Non-Dedicated Type*. We define a dedicated implementation element as a class, method or attribute completely dedicated to implement a concern.

Brinkley *et al*. [6] advocated that refactoring dedicated implementation elements should be straightforward since it only requires moving them from a class to an aspect by using inter-type declarations, for instance. However, a high number of refactoring mistakes were observed in this experiment when developers had to refactor dedicated elements. In particular, the mistakes Duplicated Crosscutting Code and Incomplete Refactoring (Section 3) are commonly related to dedicated implementation elements. The synergy between these two mistakes and dedicated implementation elements could be due to the concern mapping task, i.e., the developer fails to assign dedicated elements to a concern. In fact, Nunes *et al*. [25] have pointed out that not assigning a dedicated implementation element to a concern is a common mistake when developers perform concern mapping. Therefore, it is not a surprise that developers make the same mistake when refactoring a crosscutting concern to aspects.

```
public class ATM {
    private static final int LOG= 0;
    ...
}
```

Fig. 3. Primitive Constant code pitfall

Primitive Constant. The first code pitfall, named Primitive Constant, occurs when a constant is dedicated to implement the concern. For instance, Figure 3 shows a constant called LOG that implements the Logging concern in the ATM application. In our study, 85% of the subjects of the first round and 86% of the second round made mistakes when refactoring this part of the Logging concern. For instance, subjects often replaced this constant by its value in places where it is used; this seems a common strategy to simplify the refactoring. However, they did not remove this constant

declaration from the base code which results in an unreachable code [10]. This kind of mistake was classified as Duplicated Crosscutting Code since both the base code and the aspect are dealing with the constant value. In addition to Duplicated Crosscutting Code, we observed that Primitive Constant is also related to the Incomplete Refactoring mistake. One example of this kind of mistake occurs when developers just ignored moving the constant to an aspect, leaving it at the base code.

Attribute of a Non-dedicated Type. This code pitfall has to do with the type of a dedicated implementation attribute. In this case, the attribute has either a primitive type (e.g., boolean, int, long) of a non-dedicated element type. For instance, the Timing concern in Telecom is implemented by two attributes. One of them, shown in Figure 4.a, is of the type Timer which is completely dedicated to the concern. Subjects of the experiment had no problem to locate and refactor this attribute to an aspect. On the other hand, Figure 4.b shows the `totalConnectionTime` attribute, declared as a long data type, which is not dedicated to Timing; it is actually of a primitive Java type. When this concern is refactored, 33% of subjects in the fifth replication and 32% in the sixth replication did not aspectize such attribute, but none of them failed to refactor the `timer` attribute (Figure 4.a). We observed that developers appear to have difficulty to assign attributes of non-dedicated types to a concern during concern mapping tasks. This fact is probably due to the lack of searching support from IDE. Consequently, the presence of such attributes in the concern code leads programmers to make the Incomplete Refactoring mistake.

```
public abstract class Connection {
   Timer timer = new Timer();
   ...
}
```

(a) Attribute of a dedicated type

```
public class costumer {
   ...
   long totalConnectionTime = 0;
   ...
}
```

(b) Attribute of a non-dedicated type

Fig. 4. Attribute of the Non-Dedicated Type code pitfall

4.2 Non-dedicated Implementation Elements

Many mistakes occurred when subjects had to refactor elements which are not completely dedicated to implement a concern; we call them non-dedicated implementation elements. To refactor these elements, developers usually rely on pointcut/advice constructs of AOP languages. Three code pitfalls are related to non-dedicated implementation elements: *Disjoint Inheritance Trees*, *Divergent Join Point Location*, and *Concern Attribute Accesses*. These code pitfalls are discussed below.

Disjoint Inheritance Trees. This code pitfall is characterized by two or more inheritance trees involved in the concern realization. Elements of one tree usually does not refer to the elements of the other. The lack of explicit relationships between two inheritance trees might be one reason that subjects correctly refactored elements in one tree, but not in the other. In addition, elements in the same inheritance tree tend to follow the same pattern in the concern implementation. Figure 5.a illustrates the

Disjoint Inheritance Trees code pitfall. Elements inside the dotted square are more prone to the Incomplete Refactoring mistake because they do not follow the same pattern of concern code implemented in tree leaves. This mistake was very recurring while the experiment subjects aspectized the ErrorMessages in the Chess application.

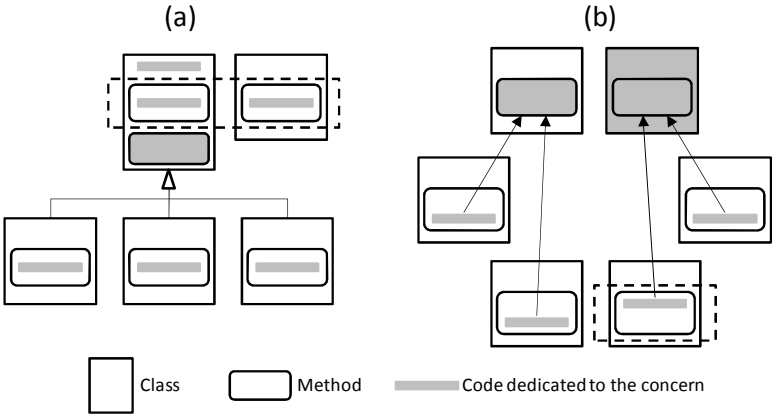


Fig. 5. Disjoint Inheritance Trees and Divergent Join Point Location code pitfalls

Divergent Joint Point Location. This code pitfall occurs when (i) pieces of the concern code are scattered over several methods and (ii) these pieces of code appear in different locations with respect to the method body. The problem mainly occurs when just a couple of pieces of code have a different location compared to the others. Figure 5.b illustrates this code pitfall. In this figure, the concern code is located at the end of three methods, but at the beginning of one method (marked with a dotted square). Therefore, developers should use after advice to refactor the former pieces of code, but a before advice in the latter case. This divergent joint point location usually tricks developers and they end up making the Incorrect Advice Type mistake.

As an example of this code pitfall, the Logging concern in the ATM application is realized by the `Logger` class and several scattered calls to the `log()` method of this class. Only one call to `log()` appears in the beginning of a method and should be refactored using a before advice. All other calls to `log()` appear in the end of a method and should be refactored using after advice. Thus, developers should use the appropriate advice type - before or after in this case - to refactor the Logging concern. We noticed that many subjects of the first and second rounds refactored all of the Logging concern parts using only after advice. They thus made the Incorrect Advice Type mistake in the case which they were expected to use a before advice. Subjects of the other four rounds have not made this type of mistake because ErrorMessages (Chess) and Timing (Telecom) have a uniform implementation. That is, all parts of the concern should be refactored using the same advice type.

Accesses to Local Variables. This code pitfall is characterized by the presence of concern code inside a method that accesses a local variable. Previous work [8] has advocated that the dependence on local variables makes the concern code harder to

refactor, since the join point model of AspectJ-like languages cannot capture information stored in such variables. On the other hand, capturing the value of class members and method parameters is straightforward. For the former, for instance, it can be done by using the *args* AspectJ pointcut designator.

```
public class ATM {
  private void performTransactions() {
    ...
    while(!userExited) {
      int mainMenuSelection = displayMainMenu();
      switch(mainMenuSelection) {
        ...
        case Log:
          Logger.printLog();
          break;
        ...
      }
    }
  }
  ...
}
```

Fig. 6. Incorrect Implementation Logic related Logging

To illustrate this code pitfall, we rely on a piece of the Logging concern (ATM) shown in Figure 6. About 40% of subjects made the Incorrect Implementation Logic mistake when refactoring to aspects this piece of code. The logical expression that controls the execution flow of the switch statement in Figure 6 is composed by the local variable `mainMenuSelection`. This variable is declared inside the method body in which this code appears. In order to aspectize the grey code in Figure 6, developers need to access to this variable and check whether its value is equal to the `LOG` constant. However, this refactoring is not easy and, so, developers recurrently make mistakes in such trick aspectization scenario.

5 Tool Support

This section presents ConcernReCS , an Eclipse plug-in to find the aspectization code pitfalls discussed in Section 4. The tool is based on static analysis of Java code and builds on knowledge of our experiment results. ConcernReCS extends ConcernMapper [29] which is a tool to allow the mapping of methods and attributes to concerns. Figure 7 describes a simplified flow chart of ConcernReCS. ConcernReCS receives information from both ConcernMapper and Eclipse. The Data Analyzer module relies on this information to generate warnings of code pitfalls.

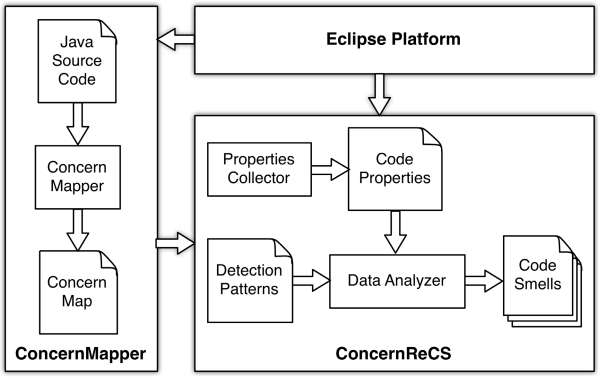


Fig. 7. ConcernReCS simplified flow chart

The first step to use ConcernReCS is to manually map the dedicated implementation elements (i.e. methods, classes and attributes) of one or more concerns using ConcernMapper. Non-dedicated implementation elements are automatically inferred by (i) calls to dedicated implementation methods, (ii) read or write accesses to dedicated implementation attributes, and (iii) syntactic references to dedicated classes or interfaces. Concern code of any other kind (e.g. conditional statements), should be extracted by the Extract Method refactoring [16] and, then, the resulting method should be mapped to the concern.

Figure 8 presents the main view of ConcernReCS in the Eclipse IDE. Code pitfalls are presented one per line and, for each of them, it is shown the code pitfall name and the mistake that it can lead to. ConcernReCS also indicates the concern in which the code pitfall appears, the source file and where in the system code it appears. For each code pitfall, the tool also gives a number representing its error-proneness. The error-proneness value can be 0.25, 0.5, 0.75, or 1, each of which representing approximately the percentage of subjects who made mistakes related to the code pitfall (the value of 1 means that all subjects in our experiment made mistakes related to a specific code pitfall).

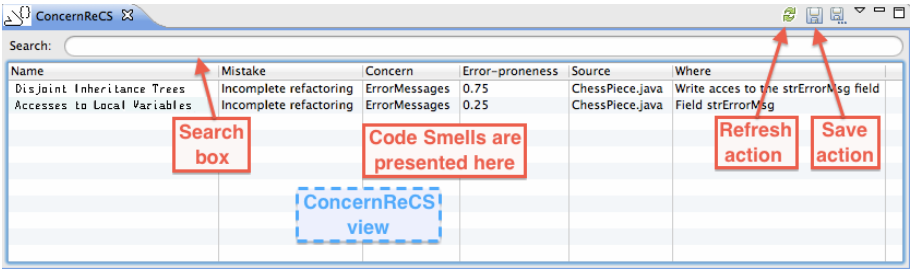


Fig. 8. The ConcernReCS main view

6 Study Limitations

The AspectJ language can be pointed out as a limitation in our study. However, it is widely known that AspectJ-based systems have been used in the large majority of AOP assessment studies. Even though there are other programming techniques which are able to realize the AOP concepts (examples are Intentional Programming, Meta Programming and Generative Programming [15]), so far AspectJ has been the mainstream AOP technology within both industrial and academic contexts [27].

Another limitation regards the size and representativeness of the target applications and the selected crosscutting concerns to be refactored. Regarding the size of the applications, they indeed do not reflect the complexity observed in the industrial context. However, this type of experiment requires tasks being performed within a short pre-specified period of time. Furthermore, applications of similar size have been used in recent programming-related experiments with different goals within varied contexts [7, 20]. The crosscutting concerns are of different nature but with similar complexity in order to pose balanced implementation scenarios in terms of difficulty.

The number and representativeness of the subjects can also be considered a limitation of this study. Around 50% of the experiment participants declared to have moderate to high level of knowledge in OOP. Despite of the observed heterogeneity, all subjects made similar types of mistakes while performing the experiment tasks. Note that their level of expertise in AOP was indeed not expected to be high, since the experiment was designed to be performed by novice programmers in AOP.

7 Related Work

Investigations of fault types and faulty scenarios in AO software: Previous studies in AOP-specific fault types and bug patterns [3, 5, 9, 11] address from the most basic concepts of AOP to advanced constructions of AspectJ-like languages. In previous research, Ferrari *et al.* [10] analyzed the existing taxonomies and catalogues and identified four main categories of faults that can be found in AO software. According to them, faults can be associated with pointcut expressions (i.e., quantification mechanisms), inter-type declarations and other declare-like expressions (i.e., introduction mechanisms), advice signatures and bodies (i.e., the crosscutting behavior), and in the intercepted (i.e., base) code. When we compare the contributions of this paper to the results presented by Ferrari *et al.* [10], we can spot one major difference. We characterize mistakes that novice programmers are likely to make while refactoring crosscutting concerns. Ferrari *et al.*, on the other hand, characterize faults that can be revealed during the testing phase, be them refactoring-related or not, which shall be prioritized in testing strategies.

Burrows *et al.* [7] analyzed how faults are introduced into existing AO systems during adaptive and perfective maintenance tasks. The AO version of the Telecom system was used as a basis for a set of maintenance tasks performed by experienced, paired programmers. The authors then applied a set of coupling and churn metrics in order to figure out the correlation between these metrics and the introduced faults.

Our study differs from the Burrows *et al*. one in the sense that while they evolved an existing AO system, our study participants were assigned to tasks of refactoring crosscutting concerns of OO systems in order to produce equivalent AO counterparts.

Previous studies about AOP-specific mistakes [3, 9, 10, 11] range from the most basic concepts of AOP to advanced constructions of AspectJ-like languages, such as intertype declarations. Since our study was conducted only with novice programmers in AOP, our classification partially overlaps previous fault taxonomies and adds new categories, such as compilation errors or warning and incomplete refactoring. Moreover, it focuses on mistakes made by programmers when using the two most basic features of AOP languages: pointcut and advice.

Investigations of concern identification and code smells: Recent studies [12, 25] investigated mistakes in the projection of concerns on code – a key task in concern refactoring. For instance, Figueiredo *et al*. [12] noticed that programmers are conservative when projecting concerns, i.e. they make omissions of concern-to-elements assignments. Such false negatives may lead to mistakes like Incomplete Refactoring or even Incorrect Implementation Logic herein described. Nunes *et al*. [25] characterized a set of recurring concern mapping mistakes made by software developers. Amongst the causes that led to mistakes in the mapping tasks, Nunes *et al*. highlight the crosscutting nature – i.e., spreading or tangling – of several concerns they analyzed. This may help us to explain the high number of refactoring mistakes observed in our study: the participants had to first identify which parts of the code referred to the target crosscutting concern. That is, they had to map the target concern in the code. Then, they should perform the refactoring step.

Macia *et al*. [21] report on the results of an exploratory study of code smells in evolving AO systems. They analyzed 18 releases of three medium-sized AO systems to assess previously documented AOP code smells and new ones characterized by the authors. Although our goal in this paper was neither characterizing nor assessing code smells for AO (AspectJ) programs, we shall perform similar analysis of the code pitfalls described in Section 4. For instance, we can investigate the relationships between different pitfalls spotted in common implementations and the similarities of these pitfalls with code smells documented by previous studies.

8 Conclusions and Future Work

This paper presented the results of a series of experiments whose main goal was to characterize mistakes made by novice programmers in typical crosscutting concern refactorings. In this study, we revisit recurring mistakes made by programmers with specific backgrounds (Section 3). The results of this study may be used for several purposes, like helping in the development and improvement of new AOP languages and tools. Based on an analysis of recurring mistakes, this paper proposed a catalogue of aspectization code pitfalls (Section 4). These code pitfalls appear to lead programmers to make the documented mistakes. To support the automatic detection of code pitfalls, we developed a prototype tool called ConcernReCS (Section 5).

Further research can rely on our study settings to perform new replications of the experiment in order to (i) enlarge our dataset, (ii) uncover additional mistakes, and (iii) expand the proposed catalogue of code pitfalls. In fact, we have plans to replicate ourselves this study using different applications and subjects. This shall allow us to perform more comprehensive analysis with statistical significance. Moreover, we aim to evaluate whether the proposed tool (ConcernReCS) is effective to advise programmers about code pitfalls and help them avoiding typical refactoring mistakes.

Acknowledgments. This work was partially supported by CNPq (grants Universal 485907/2013-5 and 485235/2013-7) and FAPEMIG (grants APQ-02532-12 and PPM-00382-14). Fabiano also thanks the financial support received from Federal University of São Carlos (RTN grant).

References

1. Results of Avoiding Code Pitfalls in Aspect-Oriented Programming (2014), `http://www.dcc.ufmg.br/~figueiredo/aop/mistakes`
2. AJDT: AspectJ Development Tools, `http://www.eclipse.org/ajdt/`
3. Alexander, R.T., Bieman, J.M., Andrews, A.A.: Towards the Systematic Testing of Aspect-Oriented Programs. Colorado State University, TR CS-04-105 (2004)
4. Alves, P., Santos, A., Figueiredo, E., Ferrari, F.: How do Programmers Learn AOP: An Exploratory Study of Recurring Mistakes. In: LA-WASP 2011 (2011)
5. Baekken, J., Alexander, R.: A Candidate Fault Model for AspectJ Pointcuts. In: International Symposium on Software Reliability Engineering (ISSRE), pp. 169–178 (2006)
6. Brinkley, D., Ceccato, M., Harman, M., Ricca, F., Tonella, P.: Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects. IEEE Transactions on Software Engineering (TSE) 32(17), 698–717 (2006)
7. Burrows, R., Taiani, F., Garcia, A., Ferrari, F.C.: Reasoning about Faults in Aspect-Oriented Programs: A Metrics-based Evaluation. In: ICPC, pp. 131–140 (2011)
8. Castor Filho, F., Cacho, N., Figueiredo, E., Garcia, A., Rubira, C., Amorin, J., Silva, H.: On the Modularization and Reuse of Exception Handling with Aspects. Software: Practice and Experience 39(17), 1377–1417 (2009)
9. Coelho, R., Rashid, A., Garcia, A., Ferrari, F., Cacho, N., Kulesza, U., von Staa, A., Lucena, C.: Assessing the Impact of Aspects on Exception Flows: An Exploratory Study. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 207–234. Springer, Heidelberg (2008)
10. Ferrari, F., Burrows, R., Lemos, O., Garcia, A., Maldonado, J.: Characterising Faults in Aspect-Oriented Programs: Towards Filling the Gap between Theory and Practice. In: Brazilian Symposium on Software Engineering (SBES), pp. 50–59 (2010)
11. Ferrari, F., Maldonado, J., Rashid, A.: Mutation Testing for Aspect-Oriented Programs. In: International Conference on Software Testing (ICST), pp. 52–61 (2008)
12. Figueiredo, E., Garcia, A., Maia, M., Ferreira, G., Nunes, C., Whittle, J.: On the Impact of Crosscutting Concern Projection on Code Measurement. In: AOSD, pp. 81–92 (2011)
13. Figueiredo, E., et al.: Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: Int'l Conference on Software Engineering (ICSE), pp. 261–270 (2008)
14. Figueiredo, E., et al.: On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In: CSMR, pp. 183–192 (2008)

15. Filman, R.E., Friedman, D.: Aspect-Oriented Programming is Quantification and Obliviousness. Aspect-Oriented Software Development, pp. 21–35. Addison-Wesley (2004)
16. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
17. IEEE Standard Glossary for Software Engineering Terminology. Standard 610.12, Institute of Electrical and Electronic Engineers, New York, USA (1990)
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
20. Lemos, O., Ferrari, F., Silveira, F., Garcia, A.: Development of Auxiliary Functions: Should You Be Agile? An Empirical Assessment of Pair Programming and Test-First Programming. In: ICSE, pp. 529–539 (2012)
21. Macia, I., Garcia, A., von Staa, A.: An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems. In: AOSD, pp. 203–214 (2011)
22. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice Hall (2000)
23. Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. In: AOSD (2003)
24. Monteiro, M.P., Fernandes, J.M.: Towards a Catalogue of Refactorings and Code Smells for AspectJ. In: Rashid, A., Akşit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 214–258. Springer, Heidelberg (2006)
25. Nunes, C., Garcia, A., Figueiredo, E., Lucena, C.: Revealing Mistakes in Concern Mapping Tasks: An Experimental Evaluation. In: CSMR, pp. 101–110 (2011)
26. Piveta, E., Hecht, M., Pimenta, M., Price, R.: Detecting Bad Smells in AspectJ. Journal of Universal Computer Science 12(7), 811–827 (2006)
27. Rashid, A., et al.: Aspect-Oriented Programming in Practice: Tales from AOSE-Europe. IEEE Computer 43(2), 19–26 (2010)
28. Robillard, M., Murphy, G.: Representing Concerns in Source Code. ACM Transactions on Software Engineering and Methodology (TOSEM) 16, 1 (2007)
29. Robillard, M.P., Weigand-Warr, F.: ConcernMapper: Simple View-based Separation of Scattered Concerns. In: OOPSLA Workshops, pp. 65–69 (2005)
30. Soloway, E., Ehrlich, K.: Empirical Studies of Programming Knowledge. IEEE Transactions on Software Engineering (TSE), 595–609 (1984)
31. Spohrer, J., Soloway, E.: Novice Mistakes: Are the folk Wisdoms Correct? Communications of the ACM 29(7) (1986)
32. Munhoz, F., et al.: Inquiring the Usage of Aspect-Oriented Programming: An Empirical Study. In: International Conference on Software Maintenance (ICSM), pp. 137–146 (2009)

Bounds Check Hoisting for AddressSanitizer

Simon Moll[1], Henrique Nazaré[2], Gustavo Vieira Machado[2], and Raphael Ernani Rodrigues[2]

[1] Saarland University, Germany
s9simoll@stud.uni-saarland.de
[2] Universidade Federal de Minas Gerais, Brazil
{hnsantos,raphael}@dcc.ufmg.br, gustavovm@ufmg.br

Abstract. The C programming language is not memory safe, that is to say that the semantics of out-of-bounds memory accesses are undefined. There are tools that make certain guarantees about memory safety for C programs. Amongst these are SAFECode and AddressSanitizer. The latter instruments C programs with runtime checks to guarantee that no invalid memory accesses are allowed to execute. As is to be expected, this incurs in a notable performance decrease in instrumented programs. Our work consists in hoisting these checks out of loops in such a way that we maintain AddressSanitizer's semantics, but, by providing increased locality of access and by increasing the stride of bounds checks, we make said checks notably cheaper. Unlike previous approaches to bounds check hoisting, we use a parametric interval analysis to bound the index ranges used in array accesses. We evaluated our method on a collection of benchmarks from Polybench and from the domain of scientific computing. The optimization recovers 60.6% of the overhead introduced by AddressSanitizer on average. Since energy performance is a crucial factor on mobile systems, we have also evaluated our proposed solution on embedded systems in this regard. We observed a 31.7% reduction in energy consumption in programs instrumented with AddressSanitizer.

1 Introduction

The C programming language is an imperative language that was originally designed to provide a straightforward and efficient mapping from the language constructs to machine code. The language offers programmers a way to perform low-level accesses to the computer memory in the form of pointers. Programs written in C can perform a wide range of arithmetic operations on pointers, i.e., addition and subtraction. Techniques that resort to manipulation of memory addresses may give better optimization opportunities, but may also lead to fairly complicated and, hence, error-prone code.

There are a series of problems that arise from this flexibility of manipulating memory addresses. Amongst these are uninitialized pointers, dangling pointers, and array out-of-bounds accesses. Uninitialized pointers are, as the name suggests, pointers which are used without first being initialized. If its initial value

F.M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 47–61, 2014.

contains an address such as that of a null pointer, then dereferencing it may raise an exception. If not, then we can access garbage or, possibly, valid data placed somewhere in memory. Second, dangling pointers reference objects which were once allocated, but are not longer so. That is, the pointers are not temporally valid. Dereferencing these may not classify as an invalid access, since the memory position may have been allocated to another object. Nonetheless, these accesses are not logically valid. Finally, accessing memory positions beyond the array bounds is dangerous, as well. Array out-of-bounds accesses may be exploited by attackers with a technique called *buffer overflow* [18]. These problems are specially interesting because, in addition to the issue of correctness, they also present a serious underlying security issue.

As C and C++ strive at producing fast programs, the legality of a memory access is not usually checked, unless the programmer specifies otherwise. The high-performance and the minimal requirement of runtime support make C and C++ resource-efficient and popular in embedded systems programming. Unsafe memory accesses, however, may cause data corruption or program crashes. Academia has dedicated a reasonable amount of effort in the last decades to mitigate such problems with a minimum amount of impact on the existing code bases. Common solutions try to statically prove memory safety [5] or dynamically verify the safety of accesses [19,10]. Due to the undecidability of most compiler-related problems, static analyses are necessarily incomplete and can not avoid dynamic checks entirely [6,12].

However, the instrumentation of memory accesses has a negative impact both in terms of run time and in terms of memory usage. Even state-of-the-art runtime approaches tend to slow down programs in such a manner that makes their use in real-world applications impractical. For instance, AddressSanitizer [19] was found to produce a 73% overhead, SAFECode, 67%, and SoftBound, 21%. Thus, solutions like these are commonly used for the sole purpose of software testing in order to find memory bugs in programs. The actual goal of this research is not to develop tools to aid program testing. Instead, the goal is to create a sound and scalable strategy to produce memory-safe C and C++ programs that is compatible with already existing programs. Thus, the optimization of these strategies is an important problem, given that it would allow us to apply these in real-world programs.

In this paper, we propose a technique that mitigates the problems previously discussed.

We base our technique on AddressSanitizer which, while having the largest overhead amongst the aforementioned tools, is the most practical one. AddressSanitizer has been used on large scale software projects and works with either instrumented or uninstrumented external libraries. It is the only tool to track invalid pointers through casts to arbitrary types.

We have identified that most of the overhead caused by AddressSanitizer comes from array bounds checks performed inside loops. The performance loss

becomes even more evident when an array has its positions checked in nested loops. Furthermore, we have observed that in many cases the number of checks performed on an array is greater than the array size. This gives us strong evidence that AddressSanitizer is doing redundant work, checking the same memory position more than once.

Our approach consists in moving away from nested loops the code that guards arrays against invalid accesses. With this, we are able to reduce the number of times that each test is executed. Our transformation removes the bounds checks from the body of the loop and inserts code that dynamically checks the entire array before the first iteration of the outermost loop. If the dynamic test on the entire array succeeds, the program executes the loop without checks. Otherwise, if the test fails, the program executes an instrumented version of the loop, in order to preserve AddressSanitizer's semantics. Our approach can be applied on general applications, but it really shines when used on affine loops. Such loops are common in high performance computing, typically used in physics, biochemistry, linear algebra, etc.

Figure 1 shows an example of our optimization. Figure 1a contains the original, unchecked implementation of a simple sorting algorithm. This program performs $O(n^2)$ accesses to the array `A`, that has size n. Figure 1b shows the same program, instrumented with AddressSanitizer's checks. Figure 1c shows the program after our optimization. Thus, we remove the array checks inside the loops and check the entire array once, before the first iteration of the loop. If the test succeeds, we execute the unchecked loops. This, in practice, reduces the number of checks from $O(n^2)$ to n, speeding up the execution of the checked program. On other hand, if the tests fail, we execute the checked version of the loop. In this case, we have a performance penalty, because we check the array more times than AddressSanitizer originally would have.

We have implemented a prototype of our optimization in the LLVM compiler infrastructure [8]. We have also conducted experiments to evaluate the impact of our solution in benchmarks from the Easybench[1] and Polybench [16] suites. We have observed an average speedup of 39.8% (geometric mean) in the instrumented benchmarks, compared to AddressSanitizer. Our technique complements static approaches such as that of Nazaré *et al.* [12], that statically proves the safety of memory accesses throughout the program. Our optimization pushes AddressSanitizer a step further towards the goal of implementing scalable memory safety for C and C++ programs.

The rest of this paper is organized as follows: Section 2 gives us the basis upon which we develop our technique. Section 3 describes our algorithm and explains our engineering choices. We experimentally evaluate the performance of our work in Section 4. In Section 5 we discuss how our work is related with existing efforts in the literature. Finally, in Section 6 we discuss possible future directions of this research and make final remarks.

[1] `http://code.google.com/p/easybench-suite/`

```
char A[n];
for (int i=0; i<n; i++) {
  for(int j=i+1; j<n; j++) {
    x = A[i];
    y = A[j]
    if (x > y) {
      A[i] = y;
      A[j] = x;
    }
  }
}
```

(a) Original program

```
char A[n];
for (int i=0; i<n; i++) {
  for(int j=i+1; j<n; j++) {
    asan_check_byte(A,i);
    x = A[i];
    asan_check_byte(A,j);
    y = A[j]
    if (x > y) {
      asan_check_byte(A,i);
      A[i] = y;
      asan_check_byte(A,j);
      A[j] = x;
    }
  }
}
```

(b) AddressSanitizer-instrumented program

```
char A[n];
if (check_range(&A[0],&A[n-1])) {
  for (int i=0; i<n; i++) {
    for(int j=i+1; j<n; j++) {
      x = A[i];
      y = A[j]
      if (x > y) {
        A[i] = y;
        A[j] = x;
      }
    }
  }
} else {
  for (int i=0; i<n; i++) {
    for(int j=i+1; j<n; j++) {
      asan_check_byte(A,i);
      x = A[i];
      asan_check_byte(A,j);
      y = A[j]
      if (x > y) {
        asan_check_byte(A,i);
        A[i] = y;
        asan_check_byte(A,j);
        A[j] = x;
      }
    }
  }
}
```

(c) Program after the hoisting of AddressSanitizer's bounds checks

Fig. 1. Motivating example

2 Background

2.1 C, C++ and Safety

C and C++ programs are vulnerable to a series of runtime errors which, in many ways, compromise their safety. The latter has addressed a few of these problems and continues to do so with every new version of its standard. The recent introduction of the `std::shared_ptr` and `std::unique_ptr` smart pointers, for example, provide a way of overcoming dangling pointer and memory leak problems that exist in the core of C++, as well as that of C. This core language is weakly typed, which, naturally, leads to very few guarantees being made regarding memory safety [15]. The semantics of accessing an invalid memory position, for example, are undefined. This means that, upon executing an invalid access,

the program may abort, continue, or perform some other implementation-defined operation. Perhaps due to performance considerations, in nearly all implementations of the C and C++ standards, the program simply continues execution, albeit in an undefined state.

This unsafe nature has, historically, been the cause of several well-known and disastrous problems. The 1988 Morris worm exploited a buffer overflow vulnerability in the `finger` daemon to spread itself throughout the Internet [7], causing an estimated $100,000–$10,000,000 in damages. A similar vulnerability was recently discovered in OpenSSL.

> The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).[2]

This Heartbleed bug exploited an out-of-bounds read to allow well-crafted input to read a portion of memory, which could possibly contain sensitive information. This bug was said to have affected over two-thirds[3] of all web servers and has an estimated impact in the order of millions of dollars[4].

To detect these memory-related problems, we can resort to verification software employing static analysis. For some programs, however, static analysis alone is not a viable option, be it for the high run times or the lack of precision. In these cases, we can resort to statically instrumenting programs to make additional guarantees about them. SAFECode [6], for example, modifies the program to guarantee correct aliasing at runtime. That is, if any two pointers are said not to alias, the program will reach no state in which they do. Valgrind [13] also instruments binaries, to ensure, amongst other things, that no out-of-bounds access is allowed to execute. AddressSanitizer [19] does the same as Valgrind, but rather instruments the program at compile time. AddressSanitizer's approach, however, is heuristic and the bounds checks can very rarely incorrectly allow invalid access to execute. These tools come with an associated runtime overhead. Reported overheads are 67% for SAFECode on the Olden benchmark suite and 73% for AddressSanitizer on the SPEC CPU2006 benchmark suite. Nethercote *et al.* [13] have measured the overhead of Valgrind on the SPEC CPU2006 benchmark suite to be 2220% with memory access validation enabled. We chose to apply our research to AddressSanitizer due to its high quality, wide-spread usage, and integration with the LLVM [8] framework and its C and C++ frontend. Our techniques, can, however, be applied to other tools such as the other aforementioned.

In the section that follows, we will give an in-depth explanation of the AddressSanitizer internals.

[2] `http://heartbleed.com`

[3] `http://edition.cnn.com/2014/04/08/tech/web/heartbleed-openssl/`

[4] `http://www.theaustralian.com.au/business/latest/heartbleed-fix-to-cost-millions/story-e6frg90f-1226893225719`

2.2 AddressSanitizer

AddressSanitizer instruments the input program at compile time. This instrumentation serves two purposes, to check the *shadow state* and to create *poisoned redzones* around stack allocations and global objects, both done to detect out-of-bounds accesses. AddressSanitizer also redirects calls to `malloc`, `free` and similar functions from the runtime library to its own corresponding versions, which create poisoned redzones around objects in the heap and quarantine freed memory regions. We will explain these below.

AddressSanitizer shadows the heap into a *shadow memory*. This shadow memory maps each byte in the heap to a few corresponding bits. This by default is an 8-1 mapping, meaning each 8 bytes of the heap are mapped into 1 byte in the shadow memory. This byte's address is given by `(Addr >> 3) + Offset`, in which `Offset` is the start of the shadow memory and `Addr` is the address being accessed. The content of this byte is called the *shadow state*. If the shadow state is not zero, then an invalid - quarantined or unallocated - memory position is being accessed. This check is:

```
ShadowAddr = (Addr >> 3) + Offset;
if (*ShadowAddr != 0)
  ReportAndCrash(Addr);
```

When we refer to bounds checks in AddressSanitizer, we are referring to the above sanity check. The shadow memory is modified accordingly by the allocation and deallocation functions.

Poisoned redzones are regions of memory created around stack, global, and heap objects which are considered unaccessible and, thus any attempt to access them is considered invalid and will abort the program. Note that this is a heuristic approach, since it is, indeed, possible to index a pointer in such a way that the resulting access skips the redzone entirely, while still indexing a portion of valid memory.

We exemplify this behavior in Figure 2. We show a function called `copy_and_print` in Figure 2a and what its AddressSanitizer-instrumented equivalent may look like in Figure 2b. The function `asan_alloc_shadow` in figure 2c shows the handling of shadow memory for memory-allocating operations. In the same figure, `asan_check_byte` shows how shadow memory is verified regarding allocation. In Figure 3, we show a possible configuration of program memory and of shadow memory. The dark bits of the program memory show in which positions accesses are valid, and the dark bits of the shadow memory shows their corresponding bytes in said memory. The lighter bits are redzones or a redzone's corresponding shadow memory.

AddressSanitizer has a runtime overhead of creating and checking the shadow memory. Since we deal only with bounds check elimination, we have an unremovable overhead of approximately 10.5%, which is that of bookkeeping, that is updating the shadow memory alone.

```
void copy_and_print(char v[], int n) {
  char buf[SIZE];
  for (int i = 0; i < n; ++i) {
    buf[i] = v[i];
    printf("%d\n", buf[i]);
  }
}
```

(a) Original `copy_and_print` function

```
void copy_and_print(char arg[], int n) {
  char buf[SIZE];
  asan_alloc_shadow(buf, SIZE);
  for (int i = 0; i < n; ++i) {
    asan_check_byte(buf, i);
    asan_check_byte(arg, i);
    buf[i] = arg[i];
    asan_check_byte(buf, i);
    printf("%d\n", buf[i]);
  }
}
```

(b) Instrumented `copy_and_print` function

```
void asan_alloc_shadow(void *buffer,
                       size_t size) {
  void *shadow = buffer >> 3 + OFFSET;
  char mask = 0x1;
  for (unsigned i = 0; i < size; ++i) {
    // Move to the next byte.
    if (!mask) {
      mask = 0x1;
      shadow++;
    }
    // Set the shadow bit.
    *shadow |= mask;
    // Move to the next bit.
    mask >> 1;
  }
}
void asan_check_byte(void *buffer,
                     unsigned idx) {
  void *shadow = buffer >> 3 + OFFSET;
  if (!(*shadow[idx/8] & (1 >> (idx % 8)))) {
    // Corresponding bit of the corresponding
    // shadow byte is not set - the memory is
    // not allocated.
    crash();
  }
}
```

(c) Example shadow allocation and shadow checking functions, similar to those of AddressSanitizer

Fig. 2. Code examples for AddressSanitizer functions and instrumented programs

3 Methodology

We have implemented our optimization as an extension of AddressSanitizer in the LLVM compiler infrastructure.

The optimization proceeds through the following steps. Firstly, it determines for every loop the ranges of its iteration variables and pointers (described in Section 3.1). Secondly, looking at each loop individually, it groups memory accesses together and merges the pointer ranges (described in Section 3.2). Thirdly, the optimization inserts sanity checks for entire pointer ranges and creates an unchecked code path (described in Section 3.3). If all range checks succeed, the program executes the loop without any checks on those pointers. Otherwise, execution continues on a fully instrumented loop. In the next paragraphs, we will describe all steps in detail referring to our running example of Figure 1a as necessary.

3.1 Iteration Variable Inference

Our method relies on the inference of the ranges of iteration variables. Iteration variables are initialized before entering a loop, change in every iteration, and are checked to decide if the loop should be terminated. Some examples can be seen in Figure 4 or our running example in Figure 1a. If we know the range of an

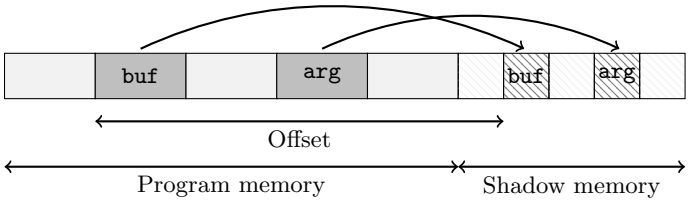


Fig. 3. Memory configuration for the program in Figure 2b

```
for (int i = 0; i < n; ++i) {
  ...
}
```

```
int j = 2 * n + 42;
do {
  ...
  j = j - 5;
  ...
} while (j > 3);
```

Fig. 4. Some examples of loops with iteration variables that our analysis can recognize. Here, `i` and `j` are iteration variables wherever they occur.

iteration variable, we can determine the range of array accesses that depend on it. Therefore, we only consider loops for which unique iteration variables can be found. For the purpose of this paper, we assume that the iteration variable and the loop it controls have the following properties:

- The iteration variable has integer type.
- In each iteration, the variable is incremented/decremented by a constant.
- Its loop has a single exit condition.
- The exit condition compares the iteration variable against a loop invariant value.

If a loop satisfies these properties our analysis infers expressions for the lower and upper bound of the iteration variable. These bound expressions are based on loop invariant program variables and constants. Consider for example, the iteration variables `i` and `j` in our guiding example of Figure 1a. The analysis infers the range $[i+1, .., n-1]$ for variable `j` and the range $[0, .., n-1]$ for variable `i`. By using interval arithmetic, the analysis can also determine ranges of values derived from the iteration variable. Note, that the bound expressions for variable `j` are loop invariant with respect to the inner loop. However, the lower bound still depends on `i`. Based on this range, any range check depending on `j` could not be hoisted beyond the outer loop. To alleviate this problem, we infer the ranges of the current bound expressions of the variable `j` with respect to the outer loop. This yields the range $[1, .., n-1]$ for the lower bound expression $i+1$. If an array access in the loop depends on `j` or `i`, we can bound its pointer range even before the loop. This is the crucial requirement for hoisting the sanity check out of the loop nest, which makes our optimization effective. Given the simple structure of the instrumented loops, the question arises why runtime checking is necessary

in these cases. Where safety of access can be proven at compile time, runtime checks can be omitted. These guarantees require an unambiguous mapping from each pointer variable to memory allocations of known size. This is impossible if only parts of the program are known at compile time as it is the case with external libraries or compilation of separate modules. Even if the whole program is known the problem is undecidable in general.

3.2 Memory Access Decomposition

As discussed in Section 3.1, we can determine loop invariant bounds for iteration variables and pointers derived from them. We decompose the accessed pointers into loop-invariant base pointers and loop-carried offsets. This corresponds to the notion of an array access. We determine the ranges of all offsets and merge those indexing the same array. This yields for each loop and array, an over-approximation of the range of indices that will be accessed. Function parameters, loaded and returned values are considered to point to distinct arrays. This gives us for each loop nest a set of arrays, each associated with a set of memory instructions and their merged ranges. This is the information needed to perform the actual hoisting explained below.

3.3 Hoisting

During hoisting, the optimization clones loop nests which contain memory accesses with known ranges. Both loop nests will be instrumented by AddressSanitizer. One version will not perform sanity checks for accesses that will be subsumed by a range check. Finally, the optimization materializes the range checks. With the default redzone size of 32 bytes, the redzones marking the ends of allocated memory extend 32 bytes from both ends of the buffer. To verify a memory range, the range check can stride at 32 bytes through the range of the memory accesses. The range check is implemented as follows:

```
bool check_range(char * low, char * high) {
  for (char * p = low; p < high; p = p + redzone_size) {
    if (! safe(p)) return false;
  }
  return safe(high);
}
```

The decision to inline the range check is left to the compiler. The array elements in our benchmarks have a size of 4 or 8 bytes, typical of numeric data types. This can make range checks faster, even if only a single element is accessed per iteration of the instrumented loop. The result of this applied to our guiding example can be seen in Figure 1c. The executed code corresponds to that found in Figure 1a for the *uninstrumented* case and to Figure 1b for the *checked* case.

4 Experiments

We integrated our optimization into the implementation of AddressSanitizer in the LLVM project. The prototype was able to automatically optimize the functions in the reported benchmarks. The measured run times are shown in Figure 5. We verified that the range checks never caused false positives. Only the unchecked loops were executed where the optimization applied. Therefore, the results reflect the pure overhead of the range checks. We provide a complementary table with code statistics in Figure 7.

The optimization is applicable in all benchmarks but *partitionStrSearch*. In the case of the *KNN* and *easy_add* the optimization only added instructions without any performance gain. We observe significant speedups for the other benchmarks.

Our results back up the claim earlier made that most part of the instrumentation overhead stems from bounds checks in loops. We conclude that bounds check hoisting can be very effective in compensating the overhead of AddressSanitizer in loop nests. The clear separation of the results points to a potential for a heuristic approach. The *easy_add* benchmark does not benefit from the optimization. Normally instrumented *kMeans* has little overhead already. However, in said benchmark the optimization blows up code size by 85%. This favors a compile-time approach to trade off expected performance gains against code size. We did not pursue this idea yet.

4.1 Energy Measurements

To determine the energy consumed in the embedded system by a specific program, we created a simple and robust procedure. To do so, a DAQ (NI USB-6009) [11] measures the instantaneous current between the load and the ground and sends it to a software in a PC called *CMeasure*. Since the tension in the embedded system is constant, this software is able to calculate the instantaneous power and, by integrating it, the consumed energy.

An application *(SET_UP_DOWN)* runs on the embedded system and notifies the software when to start and stop getting data from the DAQ. It does that by sending a signal through the RTS (Request To Send) pin from the serial port DB9.

CMeasure and *SET_UP_DOWN* were developed by us. A shunt resistor is used as the current sensor, and due to project restrictions (medium/low current system), the Low-Side Current sensing technique was used. We have also developed a series of MATLAB functions to manipulate the measured data.

Methodology. The first thing to do is to determine the basal energy consumption of the embedded system. Therefore, we execute *SET_UP_DOWN* with no application running. *CMeasure* gets all the instantaneous current values in a specific sample rate, converts it to instantaneous power, and saves it on a text file. We do it several times to get a confidence interval. Then, we use the functions we developed in a numerical computational software (*MATLAB* or *Scilab*) to manipulate this data.

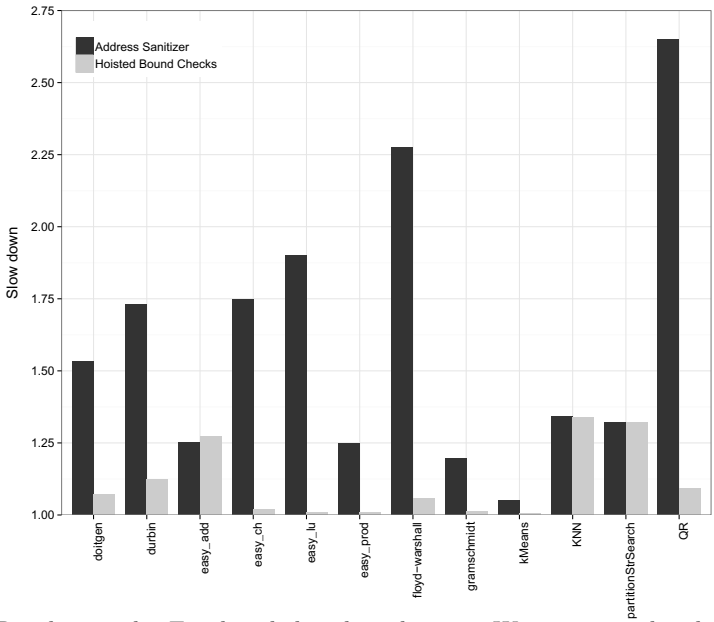


Fig. 5. Results on the Easybench benchmark suite. We compare the slow down of benchmarks instrumented by AddressSanitizer with the optimization (Hoisted Bound Checks) and without (AddressSanitizer). All experiments were conducted on a 3.4 GHz Core i7-3770 machine.

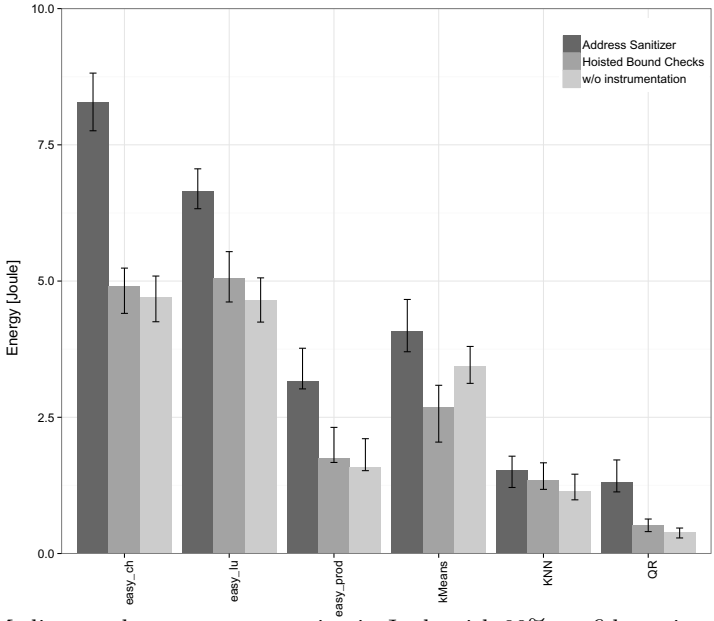


Fig. 6. Median total energy consumption in Joule with 90% confidence intervals. Measurements were taken on a *NORCO BIS - 6630* industry computer with a 1.86 GHz N2800 Intel Atom CPU.

Name	# I before	# I after	# Loops	# L opt	# Range checks
Easybench					
QR	1311	1728 (131.81%)	46	21 (45.65%)	16
easy_lu	1048	1199 (114.41%)	9	4 (44.44%)	3
easy_add	974	1081 (110.99%)	6	2 (33.33%)	3
kMeans	385	716 (185.97%)	13	11 (84.62%)	8
easy_prod	1006	1119 (111.23%)	8	3 (37.50%)	3
easy_ch	1046	1258 (120.27%)	10	3 (30.00%)	2
partitionStrSearch	244	244 (100.00%)	7	0 (0.00%)	0
KNN	411	499 (121.41%)	8	3 (37.50%)	6
Polybench					
doitgen	187	358 (191.44%)	16	16 (100.00%)	6
durbin	159	310 (194.97%)	7	7 (100.00%)	13
floyd-warshall	86	155 (180.23%)	7	7 (100.00%)	3
gramschmidt	246	477 (193.90%)	19	19 (100.00%)	9
average	-	- (141.7%)	-	- (57.8%)	-

Fig. 7. Instruction counts before and after inserting range checks and cloning the loop nests. *# L opt* is referring to the number of loops that were successfully instrumented (two nested loops count as two) with *# Range checks* many range checks in total. AddressSanitizer inserts additional code for the instrumentation which is not included here.

After we determine the basal energy consumption, we proceed to measure the system with the desired application running. Instead of executing *SET_UP_DOWN* alone, we execute it in parallel with the application. Other than this step, we repeat the steps taken in the previous procedure. It must be highlighted that, since *SET_UP_DOWN* sets the RTS to up and down in a fixed time interval, this interval must be greater than the execution time of the program. This interval has the same duration as the interval used in the basal measurement. Once we determine both energy consumptions, we subtract one value from the other to get the true energy consumption of the application.

5 Related Work

Astrée [5] is a static analyzer based on abstract interpretation [4], aimed at proving absence of runtime errors in C programs. It is able to analyze complex memory usage patterns, much more so than our work. Astrée, in fact, is sound, and considers all possible run-time errors, while generating no false positives. It was used to prove the correctness of flight control software for Airbus in 2003 and 2004, as well as that of the Jules Vernes Automated Transfer Vehicle (ATV) in 2008[5].

[5] `http://www.astree.ens.fr`

Logozzo and Fandrich [9] introduce the pentagon numerical domain used for verifying the correctness of array accesses. This domain represents properties of the form $x \in [a, b] \wedge x < y$. The goal of this is to be lightweight, more so than the octagon domain, but precise, more so than the interval domain. Their techniques were used to validate accesses in MSIL, an intermediate language for the .NET framework. They were able to verify the correctness of 83% of accesses in the `mscorelib.dll` library under 8 minutes.

The approach of Bodik *et al.* [2] to memory access validation is similar to that of Logozzo and Fandrich, but, their approach lacks information regarding intervals. They infer properties of the form $x < y$ and store this information in a sparse manner, using a newly introduced representation called extended SSA form. This representation renames variables after conditional branches in which their abstract state may change. This approach was implemented in the Jalapeño optimizing compiler infrastructure [3] and was able to remove 45% of bounds checks from selected SPECjvm98 benchmarks.

Rugina *et al.* [17] present a novel framework which is able to symbolically analyze pointer bounds and array indices, thus being able to validate array accesses. All analyses are formulated as constraint systems which are then solved with linear programming techniques. With this, they were able to verify complete correctness with regards to memory accesses on a series of artificial benchmarks.

Akritidis *et al.* [1] presented and implemented an optimization very similar to ours. They keep an uninstrumented version of a loop, as well as an instrumented version. A choice is made at runtime regarding which loop should be taken, based on whether or not the accesses can be proven safe. They implemented their algorithm on the Microsoft Phoenix code generation framework[6], and reported an 8% overhead for their entire approach, including the bounds checking portion. Their approach, unlike the one taken by AddressSanitizer, is not able to handle integer-pointer conversions, as well as not addressing the problem of temporal memory safety.

Code duplication techniques have also been used by Noorman *et al.* [14]. They duplicate functions and eliminate return instructions to protect against buffer overflow vulnerabilities. That is, instead of protecting return addresses, they are completely removed from the program. Tests on the RIPE benchmark suite, for quantifying protection of any given countermeasure, show that their approach protects against certain buffer overflows which canaries alone do not. We note that, while AddressSanitizer also provides protection against buffer overflows, it also does so with other major security vulnerabilities, including any out-of-bounds access.

6 Final Considerations

Conclusion. In this paper, we have proposed a new strategy to avoid the overhead caused by redundant memory checks. We have implemented our algorithm

[6] `http://connect.microsoft.com/Phoenix`

as an optimization to AddressSanitizer. Our heuristic was inspired by speculative approaches that are common in in just-in-time optimizing compilers. Thus, we move array bounds checks from the loop body to before the loop, where the whole array is checked only once. In order to preserve AddressSanitizer semantics, we have need of cloning loop nests, so we can run the fully instrumented version of the code when something goes wrong. Thus, the main drawback of our approach is a significant increase in program size. This increase was measured to be 41.8% on average, which still might be limiting factor for deployment in embedded systems. However, the increase in memory consumption is largely due to the shadow memory and the red zones, which are unaffected by our optimization. Even when taking into account the doubled code size in the worst case, our technique does not change total memory consumption in a meaningful way. Thus, coupled with the energy savings of 31.7%, our technique could be viable for mobile computing. Our experiments also show that our technique was able to speed up fully instrumented programs by 39.8%. If memory is not a heavily constraining factor, our optimization is, indeed, worthwhile.

Future Works. There is still room for improvements in our endeavor to make memory-safe C and C++ programs scalable. We are currently working on an extension to this research. For instance, we might evaluate how our technique performs together with other static approaches like that shown in Nazaré *et al.*. The new phase of this battle for scalability uses the idea proposed by this paper, but involves inter-procedural static analysis with partial context sensitivity and function cloning. The expected results are promising and might even make memory-safe production C and C++ programs a possibility. Our ultimate goal is to make memory-safe C code run faster than equivalent programs coded in Java, that implements memory checks by default.

References

1. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the 18th Conference on USENIX Security Symposium, SSYM 2009, pp. 51–66. USENIX Association, Berkeley (2009), http://dl.acm.org/citation.cfm?id=1855768.1855772
2. Bodik, R., Gupta, R., Sarkar, V.: ABCD: eliminating array bounds checks on demand. In: PLDI, pp. 321–333. ACM (2000)
3. Burke, M.G., Choi, J.D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M.J., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The jalapeno dynamic optimizing compiler for java. In: Proceedings of the ACM 1999 Conference on Java Grande, JAVA 1999, pp. 129–141. ACM, New York (1999), http://doi.acm.org/10.1145/304065.304113
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM (1977)
5. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)

6. Dhurjati, D., Kowshik, S., Adve, V.: Safecode: enforcing alias analysis for weakly typed languages. In: PLDI 2006: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 144–157. ACM, New York (2006)
7. Eichin, M.W., Rochlis, J.A.: With microscope and tweezers: An analysis of the internet virus of november 1988. In: Proceedings of 1989 IEEE Symposium on Research in Security and Privacy (1988)
8. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO, pp. 75–88. IEEE (2004)
9. Logozzo, F., Fähndrich, M.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. Sci. Comput. Program. 75(9), 796–807 (2010)
10. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: Softbound: Highly compatible and complete spatial safety for C. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (June 2009)
11. National Instruments Corporation: User Guide and Specifications NI USB-6008/6009: Bus-powered multifunction DAQ USB device (2004)
12. Nazaré, H., Maffra, I., Santos, W., Barbosa, L., Pereira, F., Gonnord, L.: Validation of memory accesses through symbolic analyses. In: Proceedings of the 2014 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014. ACM (to appear, 2014), Invited paper with publication expected for 2014
13. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42(6), 89–100 (2007), http://doi.acm.org/10.1145/1273442.1250746
14. Noorman, J., Nikiforakis, N., Piessens, F.: There is safety in numbers: Preventing control-flow hijacking by duplication. In: Jøsang, A., Carlsson, B. (eds.) NordSec 2012. LNCS, vol. 7617, pp. 105–120. Springer, Heidelberg (2012)
15. Pearce, D.J., Kelly, P.H., Hankin, C.: Efficient field-sensitive pointer analysis of C. ACM Transactions on Programming Languages and Systems (TOPLAS) 30(1), 4 (2007)
16. Pouchet, L.N.: PolyBench: The Polyhedral Benchmark suite
17. Rugina, R., Rinard, M.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. SIGPLAN Not. 35(5), 182–195 (2000)
18. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: Proceedings of the 11th Annual Network and Distributed System Security Symposium, pp. 159–169 (2004)
19. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX ATC 2012, p. 28. USENIX Association, Berkeley (2012), http://dl.acm.org/citation.cfm?id=2342821.2342849

Case of (Quite) Painless Dependently Typed Programming: Fully Certified Merge Sort in Agda

Ernesto Copello, Álvaro Tasistro, and Bruno Bianchi

Universidad ORT Uruguay
{copello,tasistro}@ort.edu.uy, bgbianchi@gmail.com
http://docentes.ort.edu.uy/perfil.jsp?docenteId=2264

Abstract. We present a full certification of merge sort in the language Agda. It features: termination warrant without explicit proof, no proof cost to ensure that the output is sorted, and a succinct proof that the output is a permutation of the input.

1 Introduction

Programming is an activity which consists in:

1. taking on a problem (technically named the *specification*) and
2. writing code to solve it.

Therefore, the delivery of every program implies an assertion, namely that the code meets the specification. For most programmers it is desirable to have access to technology that helps disclosing as many inconsistencies in such assertions as possible, as early as possible in the course of program development. Any such technology requires the declaration of at least some aspect of specification, so that it becomes possible to verify whether the code satisfies it. *Type systems* constitute a very successful technology of this kind, in which (aspects of) specifications are declared as types. *Dependent* type systems, in particular, make it possible to declare functional specifications in full detail, so that type checking entails actual logical correctness of the code. In other words, such languages can be said to feature an appealing catchword: *If it compiles, it works.* Now, the fact is that what actually compiles in these cases is not just the executable code, but this together with additional *mathematical* proof, namely a *formal proof* that the executable code matches the specification. This extra proof is necessary because it is impossible in general to automatically prove that a given executable code satisfies an arbitrary specification. It can be said that dependent type systems turn, at least in principle, programming into *fully formalized* mathematics. Actually, the languages arisen from this trend are functional programming languages, for example Agda [Nor07], Coq [Coq09] or Idris [Bra13], derived in general from (constructive) type theory, a system of logic intended for the full formalisation of (constructive) mathematics.

F.M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 62–76, 2014.

For some practitioners this situation is indeed very welcome, for it allows to enhance program construction with the composition of explicit foundation, which is in addition automatically checked for correctness. But for most programmers the cost is too expensive, in terms of instructional and actual work load. One specific difficulty with dependently typed programming concerns the need to ensure the termination of every computation, which is a requisite if the language is to feature decidable type checking and consistency as a language wherein to do mathematics as suggested above. This requirement necessarily places some restriction as to the forms of recursion allowed, which affects the simplicity of programming and increases the cost of production due to the necessity of often burdensome termination proofs.

As a consequence of this latter issue, the largest part of the functional programming community has, for most of the time, not paid serious consideration to the alleged importance of dependently typed programming languages, being satisfied with what is provided in various extensions of Hindley-Milner's type system. Now, as it happens, these extensions have, however, eventually begun to move towards incorporating dependently typed features. One first example is the *generalized algebraic data types* and, more recently, the use of Haskell's *type classes* as predicates and relations on types, giving rise to a kind of logic programming at the type level. As a consequence, some interest has awakened in the Haskell community concerning the power of the very much extended Haskell's type system for carrying out dependently typed programming.

A quite representative example of the mentioned investigations is [LM13], where the system of classes and kinds of Haskell is used to program a partially certified *merge sort* program. The result is quite satisfactory as the correctness properties considered are ensured in an almost totally *silent* way, that is, without need to mention any proof. But, at the same time, the certification is partial, only providing for the ordering of the output list.

Our investigation in this paper concerns the development of the merge sort example in Agda[1], a language *designed* to be dependently typed. We consider the problem of full certification, which means showing that the list resulting from merge sort is sorted and a permutation of the input and that the program terminates on every input. As a result, we are able to assert that:

1. Ensuring the sorted character of the output is achieved at the cost of minute proof obligations that are in all cases automatically inferred by the compilation system.
2. Ensuring that the output list is a permutation of the input requires a very low proof cost, due to a carefully chosen formalisation of the required condition.
3. Termination can be ensured syntactically by the use of Agda's *sized types*. This is quite significant, as recursion in this algorithm is general well-founded, that is, not structural, which makes it in general not checkable by a purely syntactic mechanism.

The rest of the paper begins by the latter feature above, explaining the *sized types* of Agda and showing how to use them for transforming a general

[1] Agda version 2.3.2.2 is used in this work.

well-founded form of recursion into a structural one. Next, we turn to discussing the specification of the sorting problem and the certification of merge sort with respect to it. We end up in section 4 with conclusions. As already said, the code shown is Agda and can be fully accessed at:

https://github.com/ernius/mergesort

2 Sized Types

Introduction. Consider the following versions of subtraction and quotient on natural numbers:

```
minus : ℕ → ℕ → ℕ
minus 0 y             = 0
minus x 0             = x
minus (suc x) (suc y) = minus x y


div : ℕ → ℕ → ℕ
div 0 y       = 0
div (suc x) y = suc (div (minus x y) y)
```

This definition of div is not structurally recursive. It works under some preconditions, namely $\mathsf{div}\,x\,y$ yields the quotient of x into y if $x \geqslant y$. Specifically, if $y = 0$ the computation is terminating with result x —which is rather arbitrary, but meets the classic quotient specification. It might therefore be deemed a peculiar version of the quotient operation, but it will anyhow serve our purpose of explaining how the mechanism of sized types can be used to achieve syntactic control of the well-foundedness of some recursion forms.

The reason why the recursion in div is well-founded is that the first argument decreases strictly in the recursive call. Indeed, $\mathsf{minus}\,x\,y < \mathsf{suc}\,x$ (even if $y = 0$). We can make this recursion structural by redesigning the *type* of the first argument to div so that it carries appropriate information about its value. If we could for instance express that x has i as an upper bound, then we would get that:

1. $\mathsf{suc}\,x$ has upper bound $\mathsf{suc}\,i$.
2. $\mathsf{minus}\,x\,y$ has upper bound i.

The type of x would in such case be specified as $\mathsf{NatLt}\,i$ (in words: *Natural numbers less than* i) and the type of div would be:
$(i : \mathbb{N}) \to (\mathsf{NatLt}\,i) \to \mathbb{N} \to \mathbb{N}$.
The important point is the appearance of i as a further parameter. Indeed, calling div will require to explicitly pass i, that is, to pass the upper bound to the subsequent parameter. Then the recursive equation that we are looking at will, when adequately rewritten, expect an upper bound $\mathsf{suc}\,i$ in the pattern on the left and effect the recursive call on just i on the right, so there is a

structural decrease of this parameter through the successive calls. Let us now write the details down. We start by introducing the natural numbers *with an upper bound*:

```
data NatLt : ℕ → Set where
  zero : (ι : ℕ) → NatLt (suc ι)
  succ : (ι : ℕ) → NatLt ι → NatLt (suc ι)
```

Now, to begin with, minus is rewritten as follows:

```
minus : (i : ℕ) → NatLt i → ℕ → NatLt i
minus _       x        0       = x
minus .(suc i) (zero i)  _       = zero i
minus .(suc i) (succ i x) (suc y) = upcast i (minus i x y)
```

The first equation considers the case in which the subtrahend is 0. Otherwise, we proceed by pattern matching on the minuend, which is the second parameter of this function. Now, it is a general phenomenon that pattern matching in the presence of dependent types induces patterns on parameters other than the one being considered. For instance, in the two cases of patterns of the second parameter of minus, the first parameter —i.e. the upper bound— cannot but be suc i. This *must* be specified when writing the equations. What would otherwise be a non-linear pattern (ones with repeated parameters at the left side of function's rules), becomes in Agda a so-called *dotted* pattern —used exclusively for the purpose of well-formation checking. The last equation makes use of a *casting* function upcast. Indeed, the recursive call to minus yields a result of type NatLt i, but what we need to return is one of type NatLt (suc i). So, upcast function only changes the type of the result.

The div function becomes:

```
div : (i : ℕ) → NatLt i → ℕ → ℕ
div .(suc i) (zero i) _   = zero
div .(suc i) (succ i x) y = suc (div i (minus i x y) y)
```

and, as mentioned above, as we proceed on the second argument the recursive call is *structurally* decreasing on the first one, which is an upper bound on the size of this argument.

Now, as it happens, the information about the bound of the arguments of type NatLt can actually be inferred from the corresponding constructors. This inference facility is what constitutes the *sized types* feature of Agda: There is to begin a type Size, which is similar to ℕ with a constructor ↑ for "successor" Then one can declare for example the *sized* natural numbers as follows:

```
data SNat : {ι : Size} → Set where
  zero : {ι : Size} → SNat {↑ ι}
  succ : {ι : Size} → SNat {ι} → SNat {↑ ι}
```

In Agda, curly braces around parameters indicate that these are optional or *implicit* when used as arguments, which means that they can be omitted

and shall then be inferred by the type-checker in function calls. Therefore the programmer may omit size information whenever this is unimportant. Besides, sized types admit *subtyping*, i.e. every expression of a type α of size i —say $\alpha\ \{i\}$— is also of type $\alpha\ \{\uparrow i\}$ and, transitively, of every other instance of α of greater size. Using sized naturals the final code for minus and div is the same as the one we began with, but now it passes Agda's termination check:

```
minus : {ι : Size} → SNat {ι} → ℕ → SNat {ι}
minus zero     y       = zero
minus x        zero    = x
minus (succ x) (suc y) = minus x y
  --
div : {ι : Size} → SNat {ι} → ℕ → ℕ
div (zero)   y = zero
div (succ x) y = suc (div (minus x y) y)
```

Merge sort. We shall now use sized lists to write a version of merge sort whose termination is certified by Agda in a purely syntactic manner, that is, without having to produce a termination *proof*. Merge sort works in two parts: it first splits the given list into two –which we shall call deal– and then merges the recursively sorted permutations of the latter into the final sorted list –which we shall call merge. Direct encoding of this algorithm using ordinary lists will not pass Agda's termination check, as the recursive calls take on the components of the result of the deal function, which are lists with no structural relation with the original list.

Lists can be defined in Agda in the following way:

```
data List : Set where
  []    : List
  _::_  : (x : A) (xs : List) → List
```

The type A of the members of the list is a parameter to the whole module or theory. It comes equipped with a *total* relation $\leqslant$ which is what is required to perform list sorting. In the code below we add size parameters to the previous definition. As can be seen, in Agda it is possible to (re)use the former, ordinary list constructors:

```
data ListN : {ι : Size} → Set where
  []    : {ι : Size} → ListN {↑ ι}
  _::_  : {ι : Size} → A → ListN {ι} → ListN {↑ ι}
```

In general, it is convenient to program functions converting back and forth between a type and its sized version, which is readily done. In this section we shall make use of the forgetN function going from sized to ordinary lists. This function is trivial so we omit its definition. Next we introduce the code of the deal function. Notice that the size annotations certify that the function does not increase the size of the input in any of the resulting lists:

```
deal : {ι : Size} (xs : ListN {ι}) → ListN {ι} × ListN {ι}
deal []          = ([], [])
deal (x :: [])   = (x :: [], [])
deal (x :: y :: xs) with deal xs
... | (ys, zs)   = (x :: ys, y :: zs)
```

We go immediately on to present the merge sort, so that it can be seen that the resulting code is indeed simple and quite natural:

```
mergeSort : {ι : Size} → ListN {↑ ι} → ListN
mergeSort []          = []
mergeSort (x :: [])   = x :: []
mergeSort (x :: y :: xs) with deal xs
... | (ys, zs)        = merge (mergeSort (x :: ys)) (mergeSort (y :: zs))
```

As in the div example, the recursion has become structural on the implicit size parameter. Indeed, the pattern of the recursive equation is of size at most $\uparrow(\uparrow \iota)$ whereas the parameters to the recursive calls can be calculated of size at most $\uparrow \iota$. It could be objected that a more natural code would apply deal in the third case directly to the whole list, making later use of the fact that in such case the resulting lists are both of lesser length than the original one. But this fact cannot be captured syntactically, specifically by means of the sized types, for these cannot tell variations in the behavior of the function at different inputs. In our case, that means we cannot distinguish the case in which the upper bound strictly decreases in the third rule of the definition of deal from those in which the upper bound remains unchanged as in the first two rules.

The merge function also needs to specify size information in order to automatically work out the termination, in spite of only performing structural recursive calls. The reason is that those calls are on alternate parameter places.

```
merge : {ι ι' : Size} (xs : ListN {ι}) (ys : ListN {ι'}) → ListN
merge [] l    = l
merge l []    = l
merge (x :: xs) (y :: ys)
with tot⩽ x y
... | .1 x⩽y  = x :: merge xs (y :: ys)
... | .2 y⩽x  = y :: merge (x :: xs) ys
```

The function tot⩽ is the one deciding the total relation $\leqslant$. It can be concluded that certification of termination of this algorithm is achieved at a low cost : It specifically demands no termination proof, but only size annotations and a not totally unnatural encoding of the recursion.

3 Sorting

Ordered lists. In order to get a fully certified sorting program we must ascertain that the output list is an ordered permutation of the input. We start with the

ordering issue. One method to accomplish the certification of a program with respect to a specification is simply to prove that the output satisfies the post-condition in every case in which the input satisfies the precondition. In our case, and considering only the ordering issue, this would amount to writing a function

$$\mathsf{lemma\text{-}sorted} : (\mathsf{xs} : \mathsf{ListN}) \to \mathsf{Sorted}\ (\mathsf{forgetN}\ (\mathsf{mergeSort}\ \mathsf{xs}))$$

for an adequately defined Sorted predicate on ordinary lists. A standard definition of the latter, relative to the $\leqslant$ relation assumed in the preceding section, is the following:

$$\frac{}{\mathsf{Sorted}\ []}\ [\textsc{nils}] \qquad \frac{x \in A}{\mathsf{Sorted}\ [\mathsf{x}]}\ [\textsc{singls}] \qquad \frac{x \leqslant y \qquad \mathsf{Sorted}\ (\mathsf{y} :: \mathsf{l})}{\mathsf{Sorted}\ (\mathsf{x} :: \mathsf{y} :: \mathsf{l})}\ [\textsc{conss}]$$

The corresponding Agda code is:

```
data Sorted : List → Set where
       -- ————————————
  nils   : Sorted []
       --
  singls : (x : A)
       -- ————————————
         → Sorted [x]
       --
  conss : (x y : A) (ys : List)
         → x ⩽ y → Sorted (y :: ys)
       -- ————————————————————————
         → Sorted (x :: y :: ys)
```

A very similar method of certification would be to use what can be called *conditioned* types. If the problem is that of transforming input of type α satisfying a precondition P into output of type β standing in the relation Q to the input, then the function to be programmed is one of type

$(x : \alpha)(P(x) \to (\exists y : \beta)\, Q(x, y))$.

In our case the sorting specification might be:

$(xs : List) \to (\exists ys : List)\, \mathsf{Sorted}\ ys$.

Elements of type $(\exists x : \beta)\gamma$ are *pairs* formed by an object of type β and a proof that it satisfies γ.

The difference between proving lemma-sorted and programming a function of the latter type is minor. In the second case we get a function in which executable code is interspersed with proof code whereas the first corresponds to program verification. In either case we have to develop proof code that follows closely the (recursive) structure of the program or executable code.

Using what we are calling *conditioned* types is but *one* way of encoding the specification into the output type. That approach takes at least two inductive definitions, namely the one of the output data (the β type above) and that of the correctness property (predicate or relation) that must be satisfied. Now, as it happens, the encoding in question can often be accomplished in a more direct way by giving just *one* inductive definition that represents the data of type β

that satisfy the required condition. Such definition will be called a *part* of the βs, which in Type Theory means βs exhibiting certain further feature or evidence. We call them for this reason *refined* data types or data structures.

Examples are of course the natural numbers NatLt with an upper bound or, generally, the sized types, both given in the preceding section. Notice for example that type NatLt could be used to give a finer specification of the subtraction, where we requires that the minuend not to be lesser than the subtrahend:
$(-) : (m : \mathbb{N}) \to \mathsf{NatLt}(\mathsf{suc}\, m) \to \mathbb{N}$.

One example of a refined data structures is the *ordered* lists introduced in [AMM05] and used again in [LM13]. It is the type of ordered lists whose elements are bounded above and below by given values. We give here a definition tailored to our needs, which on the one hand does not make use of an upper bound[2] and, on the other, is sized:

```
data Bound (A : Set) : Set where
  bot : Bound A
  val : A → Bound A
  --
data LeB : Bound A → Bound A → Set where
  lebx : {b : Bound A} → LeB bot b
  lexy : {a b : A} → a ⩽ b → LeB (val a) (val b)
  --
data OList : {ι : Size} → Bound A → Set where
  onil : {ι : Size} {l : Bound A}
       -- ------------------------
       → OList {↑ ι} l
       --
  :<   : {ι : Size} {l : Bound A} (x : A) {l⩽x : LeB l (val x)}
       → OList {ι} (val x)
       -- ------------------------------------------------------
       → OList {↑ ι} l
```

Observe the "cons" constructor :<. It requires first implicit arguments corresponding to the size and the lower bound to the members of the list. Next comes the head and, next to that, an implicit argument which is a proof that the lower bound given is indeed lesser than the head. Finally the tail must have the head as a lower bound, thus ensuring the sorted character of the entire list. In the implicit argument just mentioned, which stands between the head and the tail, we use the names LeB and val. The reason is connected to the use of the family of types Bound . This family of types is indexed by the type of the list's elements.

Bound type wraps up a type A together with an absolute lower bound bot. This is convenient in order to produce OLists without having to care for providing a

[2] That would be necessary if we should make use of the append operation on the ordered lists. This is not used in [LM13] either, but the authors do not simplify the definition.

precise lower bound thereof. That bot is indeed a minimum of Bound A in ensured by redefining the "order" relation with LeB data type.

Next, using the forgetO function from ordered lists to plain lists, we prove the correction of OList definition, verifying that any list with type OList is sorted.

```
lemma-sort : {l : Bound A} (xs : OList l) → Sorted (forgetO xs)
lemma-sort onil          = nils
lemma-sort (:< x onil)   = singls x
lemma-sort (:< x (:< y {lexy x⩽y} xs))
  = conss x y (forgetO xs) x⩽y (lemma-sort (:< y {lexy x⩽y} xs))
```

We next show the code of merge sort acting on sized OLists. The boxed parts constitute the additional code corresponding to proof obligations of the conditions related to the ordered character of the lists. The corresponding parameter places have been put into curly braces in an attempt to dispense with them as much as possible but, as it happens, they are actually required by Agda. In the case of merge sort the extra code is minimal, namely the instantiation of a parameter standing for a proof of inequality. The gray highlighting indicates that Agda is able to supply the required proof object automatically. Therefore, we observe that programming the new version of merge sort entails no cost due to proof.

It should also be observed that deal is without change, as it was to be expected. This is due to the simple fact that it is able to continue acting on just sized lists, not necessarily ordered.

```
mergeSort : {ι : Size} → ListN {↑ ι} → OList bot
mergeSort []          = onil
mergeSort (x :: [])   = :< x {l⩽x = lebx} onil
mergeSort (x :: y :: xs) with deal xs
... | (ys , zs)       = merge (mergeSort (x :: ys)) (mergeSort (y :: zs))
```

As to merge, we get the following new code, with additions of proof obligations boxed and grayed:

```
merge : {ι ι' : Size} {l : Bound A} → OList {ι} l → OList {ι'} l → OList l
merge onil l = l
merge l onil = l
merge (:< x {l⩽x =l⩽x} xs) (:< y {l⩽x =l⩽y} ys)
  with tot⩽ x y
... | .1 x⩽y = (:< x {l⩽x = l⩽x} (merge xs (:< y {l⩽x = lexy x⩽y} ys)))
... | .2 y⩽x = (:< y {l⩽x = l⩽y} (merge (:< x {l⩽x = lexy y⩽x} xs) ys))
```

The two first occur in patterns and are therefore not proof obligations, but just parameters that have to be made explicit. The other four are automatically

solved by Agda. Therefore, there is no cost associated to proof construction. Finally, we apply the previously presented lemma to the present merge sort algorithm in order to return an ordered list:

```
lemma-mergeSort-sorted : (xs : ListN) → Sorted (forgetO (mergeSort xs))
lemma-mergeSort-sorted = lemma-sort ∘ mergeSort
```

We observe that the certification of the sorted character of the output list is, if not totally silent as in [LM13], nearly so and, in any case, not more expensive, since the minute proofs are solved automatically by Agda.

The permutation relation. We now turn to the problem of ensuring that the output list is a permutation of the input. The point in this case is the choice of an appropriate formalisation of the specification. We begin by inductively defining a ternary relation whose instances will be written $xs/x \mapsto xs'$ and will hold when xs' is the result of removing an element x from an arbitrary position of the list xs. The following rules define this relation, establishing that a list element can be removed either from the head or from the tail of a list.

$$\frac{}{(x :: l)/x \mapsto l} \qquad \frac{l/y \mapsto l'}{(x :: l)/y \mapsto (x :: l')}$$

We next encode this relation in Agda:

```
data _/_↦_ : List A → A → List A → Set where
  removeFromHead : {x : A} {xs : List A}
                   --------------------
                 → (x :: xs) / x ↦ xs
  removeFromTail : {x y : A} {xs ys : List A} → xs / y ↦ ys
                   ------------------------------------
                 → (x :: xs) / y ↦ (x :: ys)
```

The preceding relation is used in the following standard definition of the permutation relation $\sim$, which does not need a decidable equality:

$$\frac{}{[] \sim []}\ [\sim []] \qquad \frac{l_1/x \mapsto l_3 \qquad l_2/x \mapsto l_4 \qquad l_3 \sim l_4}{l_1 \sim l_2}\ [\sim x]$$

The corresponding Agda code is:

```
data _∼_ : List A → List A → Set where
  ∼[] : [] ∼ []
        --
  ∼x : {x : A} {xs ys xs' ys' : List A}
       → (xs / x ↦ xs') → (ys / x ↦ ys') → xs' ∼ ys'
       ---------------------------------------------
       → xs ∼ ys
```

Now we face a situation similar to the one in the preceding paragraph. Indeed, in order to ensure that the merge sort returns a permutation of the input list, we could think of employing an appropriate refined type. The existence of the latter,

however, seems unlikely: we would need to define inductively the lists that are permutations of a given list, and there exists a large variety of transformations between the two related lists that do not seem able to be captured by simple constructors. We therefore choose to employ a specification with conditioned types, that is, one that produces an output consisting of both a list and a proof that it is a permutation of the input list. We know that the proof will follow the structure of the merge sort algorithm. Let us the look at this: First, deal returns the pair (ys, zs) of lists, which must be a partition of the original list xs. We might therefore think of proving that the concatenation of ys and zs is a permutation of xs, but this will introduce properties of the concatenation operation in our proof. We try to avoid such a detour by defining when a list is a permutation of a pair of lists, as follows:

```
data _∼p_ : List A → List A × List A → Set where
  ∼[]r : (xs : List A) → xs ∼p ([], xs)
         --
  ∼[]l : (xs : List A) → xs ∼p (xs, [])
         --
  ∼xr  : {x : A} {xs ys xs' zs zs' : List A}
       → (xs / x ↦ xs') → (zs / x ↦ zs') → xs' ∼p (ys, zs')
         -- ------------------------------------------------
       → xs ∼p (ys, zs)
         --
  ∼xl  : {x : A} {xs ys xs' ys' zs : List A}
       → (xs / x ↦ xs') → (ys / x ↦ ys') → xs' ∼p (ys', zs)
         -- ------------------------------------------------
       → xs ∼p (ys, zs)
```

The first two cases are trivial. The remaining two consider removing an element from the first list: then the same element has to be removed from (exactly) one of the other two lists and the lists resulting of the removals must still be in the permutation relation. We can indeed prove that a list is a permutation of a pair of lists according to this definition if and only if the list is a permutation of the concatenation of the pair of lists in question.

Now, for the purpose of certifying the merge sort algorithm we do not need so much, we can do with a coarser relation, only allowing to remove elements from the head of the lists. The relation is clearly a *sound*, although not *complete* version of the permutation relation, but this is all we need for ensuring that merge sort returns a permutation of the given input. The observation gives rise to the following definition:

```
data _∼p'_ : List A → List A × List A → Set where
  ∼[]r : (xs : List A) → xs ∼p' ([], xs)
  ∼[]l : (xs : List A) → xs ∼p' (xs, [])
  ∼xr  : {x : A} {xs ys zs : List A}
       → xs ∼p' (ys, zs)
```

```
  --  ------------------------------------------------
    → (x :: xs) ∼p' (ys, x :: zs)
∼xl  : {x : A} {xs ys zs : List A}
    → xs ∼p' (ys, zs)
  --  ------------------------------------------------
    → (x :: xs) ∼p' (x :: ys, zs)
```

The next lemma states that this restricted relation is indeed a sound version of the permutation relation, in other words, it implies that the first list is a permutation of the concatenation of each pair to which it is related:

```
lemma∼p'∼ : {xs ys zs : List} → xs ∼p' (ys, zs) → xs ∼ (ys ++ zs)
```

The latter is used for proving the following lemma, which establishes that given two elements in this restricted relation, if the their second components define a permutation component by component, then their first components are a permutation too. It is necessary to demonstrate the correctness of the algorithm later:

```
lemma∼p' : {xs ys zs ws ys' zs' : List A} → xs ∼p' (ys, zs) →
  ys ∼ ys' → zs ∼ zs' → ws ∼p' (ys', zs') →
  xs ∼ ws
```

We do not expose the code of the previous two lemmas due to space constraints.

Now we go on to certify that the pair of lists returned by deal is related to the latter's input list by the ∼p' relation. We use the forgetN and forgetNp functions to erase size information. We choose to encode the specification fully in the output type of the function just for the sake of showing in parallel the composition of the algorithm and the proof. We could as well have proven the desired condition as a property of a purely executable code.

```
deal : {ι : Size} (xs : ListN {ι}) →
  Σ (ListN {ι} × ListN {ι}) (λ p → forgetN xs ∼p' forgetNp p)
deal []                  = ([], []),
                           ∼[]r []
deal (x :: [])           = (x :: [], []),
                           ∼[]l (x :: [])
deal (x :: y :: xs) with deal xs
... | (ys, zs), xs∼ys, zs = (x :: ys, y :: zs),
                           ∼xl (∼xr xs∼ys, zs)
```

Again, because we have carefully selected the specification, we obtain an almost free proof. Proofs marked with grayed boxes were automatically solved, and only one proof must be given, corresponding to the case of a list with at least two members. What this requires is to prove that $x :: y :: xs \sim\text{p'}\ (x :: ys, y :: zs)$

if $xs \sim$p' (ys, zs), which is easily seen to follow using the rules ∼xl and ∼xr of the ∼p' relation, as is indeed exhibited in the code's last line.

Now we encode the merge function, whose type specifies that given two ordered lists it returns another ordered list together with a proof that it is a permutation of (the pair of) the two originally given ones. Again, the proofs obligations marked in gray were automatically solved by Agda. We use pair projections π.

```
merge : {ι ι' : Size} {l : Bound A} (xs : OList {ι} l) (ys : OList {ι'} l) →
        Σ (OList l) (λ zs → forgetO zs ∼p' (forgetO xs, forgetO ys))
merge onil l    = l, ∼[]r (forgetO l)
merge pl onil   = l, ∼[]l (forgetO l)
merge (:< x {l⩽x = l⩽x} xs) (:< y {l⩽x = l⩽y} ys)
with tot⩽ x y
... | ₁ x⩽y   = (:< x {l⩽x = l⩽x} zs), ∼xl hi
  where zs = π₁ (merge xs (:< y {l⩽x = lexy x⩽y} ys))
        hi = π₂ (merge xs (:< y {l⩽x = lexy x⩽y} ys))
... | ₂ y⩽x   = (:< y {l⩽x = l⩽y} ws), ∼xr hi
  where ws = π₁ (merge (:< x {l⩽x = lexy y⩽x} xs) ys)
        hi = π₂ (merge (:< x {l⩽x = lexy y⩽x} xs) ys)
```

As can be seen, only two proofs had to be provided, corresponding to the base cases. We end up giving the fully certified merge sort algorithm. Its type is a specification in the form of a conditioned OList, requiring that it is a permutation of the input. The only proof that could not be automatically solved is a direct application of lemma presented above to the induction hypotheses, named $xs \sim ys, zs$ in the where clause, and the permutation results of merge and deal functions. The code could be more neat if we had chosen just to verify the algorithm instead of developing it together with its proof.

```
mergeSort : {ι : Size} (xs : ListN {↑ ι}) →
  Σ (OList bot) (λ ys → forgetN xs ∼ forgetO ys)
mergeSort []       = onil, ∼[]
mergeSort (x :: []) = :< {l = bot} x {l⩽x = lebx} onil,
                      ∼x removeFromHead removeFromHead ∼[]
mergeSort (x :: (y :: xs)) with deal xs
... | (ys, zs), xs∼ys,zs
  with mergeSort (x :: ys) | mergeSort (y :: zs)
```

```
... | ys', [x::ys∼ys] | zs', [y::zs∼zs]
  with merge ys' zs'
... | ws, [ws∼ys,zs]
                    = ws,
                      lemma∼p' (∼xl (∼xr (xs∼ys, zs)))
                               x::ys∼ys'
                               y::zs∼zs'
                               ws∼ys', zs'
```

4 Conclusions

Dependently typed programming provides a framework for certified software development, one in which what compiles works. The price to pay for its use is increased cost in code production, since not only executable but also mathematical code (formal proofs) has to be produced. One specific difficulty arises in connection to program termination and the forms of recursion allowed: Totality is required in order to feature decidable type checking and, therefore, recursion has to be restricted. Normally the restriction is to structural recursion and, therefore, explicit proofs of termination have to be given for the general forms of recursion that are usually required.

In this paper we have contributed with a novel full formal certification of merge sort that illustrates how to ease the preceding problems by using appropriate techniques and features of the languages involved. Specifically:

1. Agda's *sized types* can ensure termination of algorithms by converting some forms of general recursion into structural in a silent manner.
2. The use of *refined* types that incorporate certain semantic information in the inductive definition of appropriate (sub)classes of data can lead to almost silent certifications.
3. The cost of proofs can significantly be reduced by a *careful choice of formalisation* of the specification in question.

One purpose of ours was to compare programming in a language designed with dependent types with experiences that have recently been put forward using the nowadays very elaborated type system of Haskell. Specifically with respect to [LM13], we observe that:

1. They obtain a silent certification of merge sort with respect to the sorted character of the output list. We, on the other hand, obtain a not fully, but nearly, silent version that is nevertheless without cost due to proof construction.
2. We are able to certify the termination of our version also without cost of proof, whereas the issue is not addressed in [LM13]. The price to pay in this case reduces to encoding the algorithm in a way that might not be the first choice for every programmer but that it is nevertheless clear and quite natural.

3. We are able to provide an inexpensive certification that the output is a permutation of the input, whereas the issue is not either addressed in [LM13].

It has to be said that it looks highly unlikely that the kind of dependently typed programming developed in the mentioned work is somehow able to cope with the last two issues just mentioned.

We also believe that our code is generally more elegant and comprehensible than the one obtained by pursuing the kind of logic programming given rise to by the hacking on the workings of Haskell's type classes constraint solver, even if, as we have done in this case, programs and proofs are composed as part of the same code. Actually we are ready to sustain that dependent type systems are much easier to learn and use, and indeed more natural, than the present type system of Haskell. Therefore we also prefer to pursue the expansion of the knowledge on genuine (i.e. by design) dependently typed programming rather than in intricate encodings thereof.

Possibly the main difficulty with dependently typed programming remains the restriction as to the allowable forms of recursion imposed by the needs of decidable type checking and of consistency of the internal logic via the isomorphism of propositions-as-types. Sized types have added to the alleviation of this matter, as we expect to have shown, but, of course, the problem is not fully solvable and there will always remain the necessity of developing termination proofs. The investigation of techniques to adequately formulate problems in ways that further facilitate the development of easily proven terminating programs is a matter worth pursuing. It is nevertheless important to mention that the consistency of a language as logic and the decidability of type checking are features derived from several restrictions, like in the supported recursion schemes, that are not enjoyed by the present expansion and form of use of Haskell's system of kinds and classes.

References

AMM05. Altenkirch, T., McBride, C., McKinna, J.: Why dependent types matter (2005), `http://www.e-pig.org/downloads/ydtm.pdf`

Bra13. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming 23, 552–593 (2013)

Coq09. The coq proof assistant reference manual (2009)

LM13. Lindley, S., McBride, C.: Hasochism: The pleasure and pain of dependently typed haskell programming. SIGPLAN Not. 48(12), 81–92 (2013)

Nor07. Norell, U.: Towards a practical programming language based on dependent type theory, Ph.D. thesis (2007)

Detecting Anomalous Energy Consumption in Android Applications*

Marco Couto, Tiago Carção, Jácome Cunha,
João Paulo Fernandes, and João Saraiva

HASLab, INESC TEC, Universidade do Minho, Portugal
CIICESI, ESTGF, Instituto Politécnico do Porto, Portugal
RELEASE, Universidade da Beira Interior, Portugal
{mcouto,tiagocarcao,jacome,jpaulo,jas}@di.uminho.pt

Abstract. The use of powerful mobile devices, like smartphones, tablets and laptops, is changing the way programmers develop software. While in the past the primary goal to optimize software was the run time optimization, nowadays there is a growing awareness of the need to reduce energy consumption.

This paper presents a technique and a tool to detect anomalous energy consumption in Android applications, and to relate it directly with the source code of the application.

We propose a dynamically calibrated model for energy consumption for the Android ecosystem that supports different devices. The model is used as an API to monitor the application execution: first, we instrument the application source code so that we can relate energy consumption to the application source code; second, we use a statistical approach, based on fault-localization techniques, to localize abnormal energy consumption in the source code.

Keywords: Green Computing, Energy-aware Software, Source Code Analysis.

1 Introduction

The software engineering and programming languages research communities have developed advanced and widely-used techniques to improve both programming productivity and program performance. For example, they developed powerful type and modular systems, model-driven software development approaches, integrated development environments that, indeed, improve programming productivity. These communities are also concerned with providing efficient execution models for such programs, by using compiler-specific optimizations (like, tail

* This work is integrated in the project GreenSSCM - Green Software for Space Missions Control, a project financed by the Innovation Agency, SA, Northern Regional Operational Programme, Financial Incentive Grant Agreement under the Incentive Research and Development System, Project No. 38973. The last author is supported by CAPES through a *Programa Professor Visitante do Exterior (PVE)* grant.

F.M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 77–91, 2014.

recursion elimination), partial evaluation [1], incremental computation [2], just-in-time compilation [3], deforestation and strictification of functional programs [4–6], for example. Most of those techniques aim at improving performance by reducing both execution time and memory consumption.

While in the previous century computer users were mainly looking for fast computer software, this is nowadays changing with the advent of powerful mobile devices, like laptops, tablets and smartphones. In our mobile-device age, one of the main computing bottlenecks is energy-consumption. In fact, mobile-device manufacturers and their users are as concerned with the performance of their device as in battery consumption/lifetime.

This growing concern on energy efficiency may also be associated with the perspective of software developers [7]. Unfortunately, developing energy-aware software is a difficult task, still. While programming languages provide several compiler optimizations, memory profiler tools, benchmark and time execution monitoring frameworks, there are no equivalent tools/frameworks to profile/optimize energy consumption.

In this paper we propose a methodology, based on an initial idea by [8], to monitor and detect anomalous energy consumption for the Android ecosystem: a widely used ecosystem for mobile devices. More precisely, we aim at providing Android application developers the tool support needed to develop energy-efficient applications. We propose a three layer methodology, which also account as the contributions of this paper:

- Firstly, we introduce an algorithm/application for the dynamic calibration of such model, thus allowing the automatic calibration of the model for any Android device. Moreover, we provide an API so that the calibrated power model can be accessed by the Android application we wish to monitor in terms of energy consumption.
- Secondly, we develop an Android application that automatically instruments the source code of a given Android application the developer wishes to monitor its energy consumption. The instrumentation is performed by embedding in the source code calls to the (calibrated) power consumption model API.
- Thirdly, we use a testing framework for Android applications in order to execute the (previously compiled) instrumented application. For each execution of a test case, we collect the energy consumed. Based on the energy consumed logs we performed several static analysis to detect abnormal energy consumption.

We have implemented our methodology in two different tools: one to dynamically calibrate our Android power consumption model, using a pre-defined set of calibrating applications. The second tool is used to automatically instrument the source code of an application its developed wishes to monitor in terms of energy consumption.

This paper is organized as follows: Section 2 presents the Android power consumption model, its dynamically calibrating algorithm, and the API that makes such model a reusable API. Section 3 describes the changes made to the Android

power consumption model so it can be used to monitor power consumption at the source code level, as well as the changes that the framework does to an application source code. Section 4 introduces the framework GreenDroid, describing how it works and how it relates the previous sections. Section 5 describes the results generated by the framework. Finally sections 6 and 7 present the related work and the conclusions, respectively.

2 A Dynamic Power Consumption Model

In this section we briefly discuss the Android power consumption model presented in [9]. This is a statically calibrated model that considers the energy consumption of the main hardware components of a mobile device. Next, we extend this model in two ways: firstly, we present an algorithm for the automatic calibration of the model, so that it can be automatically ported to any Android based device (Section 2.2). Secondly, we provide an API-based implementation of the model so that it can be reused to monitor other Android applications (Section 3.1).

2.1 The Android Power Tutor Consumption Model

We know that different hardware components have different impact in a mobile device power consumption. As a consequence, an energy consumption model needs not only to consider the main hardware components of the device, but also its characteristics. Mobile devices are not different from other computer devices: they use different hardware components and computer architectures that have complete different impact in energy consumption. If we consider the CPU, different mobile devices can use very different CPU architectures (not only varying in computing power, but also, for example, in the number of CPU cores), that can also run at different frequencies. The Android ecosystem was designed to support all different mobile (and non-mobile) devices (ranging from smart-watches to TVs). As a result, a power consumption model for Android needs to consider all the main hardware components and their different states (for example, CPU frequency, percentage of use, etc).

There are several power consumption models for the Android ecosystem [9–13], that use the hardware characteristics of the device and possible states to provide a power model. Next, we briefly present the power tutor model [9]: a state-of-the-art power model for smartphones [10]. The Power Tutor [9] model currently considers six different hardware components: Display, CPU, GPS, Wi-Fi, 3G and Audio, and different states of such components, as described next.

CPU : CPU power consumption is strongly influenced by its use and frequency. The processor may run at different frequencies when it is needed, and depending on what is being done the percentage of utilization can vary between 1 and 100; There is a different coefficient of consumption for each frequency available on the processor. The consumption of this component at a specific time is calculated by multiplying the coefficient associated with the frequency in use with the percentage of utilization.

LCD: The LCD display power model considers only one state: the brightness. There is only one coefficient to be multiplied by the actual brightness level (that has 10 different levels).

GPS: This component of the power model depends on its mode (active, sleep or off). The number of available satellites or signal strength end up having little dependence on the power consumption, so the model has two power coefficients: one to use if the mode is on and another to use if the mode is sleep.

Wi-Fi: The Wi-Fi interface has four states: *low-power*, *high-power*, *low-transmit* and *high-transmit* (the last two are states that the network briefly enters when transmitting data). If the state of the Wi-Fi interface is *low-power*, the power consumption is constant (coefficient for *low-power* state), but if the state is *high-power* the power consumption depends on the number of packets transmitted/received, the uplink data rate and the uplink channel rate. The coefficient for this state is calculated taking this into account.

3G: This component of the model depends on the state it is operating, a little like the Wi-Fi component. The states are *CELL_DCH*, *CELL_FACH* and *IDLE*. The transition between states depends on data to transmit/receive and the inactivity time when in one state. There is a power coefficient for each of the states.

Audio: The audio consumption is modeled my measuring the power consumption when not in use and when an audio file is playing at different volume, but the measures indicate that the volume does not interfere with the consumption, so it was neglected. There is only one coefficient to take into account if the audio interface is being used.

Static Model Calibration. In order to determine the power consumption of each Android device's component the power model needs to be "exercised". That is to say, we need to execute programs that vary the variables of each components state (for example, by setting CPU utilization to highest and lowest values, or by configuring GPS state to extreme values by controlling activity and visibility of GPS satellites), while measuring the energy consumption of the device. By measuring the power consumption while varying the state of a component, it's possible to determine the values (coefficients) to include in a device's specific instantiation of the model.

Power Tutor, as all other similar power models, uses a static model calibration approach: the programs are executed in a specific device (which is instrumented in terms of hardware) so that an external energy monitoring device[1] is used to measure the energy consumption. Although, this approach produces a precise model for that device [9], the fact is that with the wide adoption of the Android

[1] A widely used devise is available at `http://www.msoon.com/LabEquipment/PowerMonitor`.

ecosystem makes it impossible to be widely used[2]. Indeed, the model for each specific device has to be manually calibrated!

2.2 Power Model: Dynamic Calibration

In order to be able to automatically calibrate the power consumption model of any Android device, we consider a set of training programs that exercises all components of the power model. The training programs also change (over its full range) the state of each component, while keeping the other constant. In this way, we can measure the energy consumption by that particular component in that state. To measure the energy consumption, instead of using an external monitoring device as discussed before, we consider the battery consumed while running the training applications. The Android API provides access to the battery capacity of the device, and to the (percentage) level of the battery of the devices. By monitoring the battery level before and after executing a training application, we can compute the energy consumed by that application. After collecting power traces for all hardware components, a multi-variable regression approach is used to minimize the sum of squared errors for the power coefficient. Fig. 1 shows the architecture of the dynamic calibration of the power model.

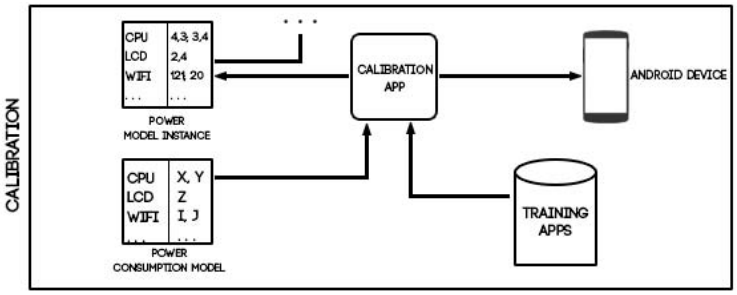


Fig. 1. The architecture to dynamically calibrate the power model for different devices

The calibration process shown in Algorithm 1 executes the set of calibration applications in a specific device. To summarize, the algorithm starts by getting the full capacity of the device's battery. Since every component has multiple possible states (e.g., CPU with different frequencies), every training program has an equal number of execution states that will be executed. Then, every state is executed **N** time, in order to get an average of the consumption. This makes the results more reliable. This algorithm returns a collection of energy consumption coefficients, one per state of every hardware component. The generic power model presented in the previous section is then instantiated.

These coefficients are used to compute the energy consumption of an Android application. For example, when the CPU component is in a known state (i.e.,

[2] In fact, [9] reports the calibration of the power model for three devices, only.

Algorithm 1. Calculate power model coefficients

```
N ← 20
capacity ← getBatteryCapacity();
for all prog : trainingProgsSet do
    for all state : getStates(prog) do
        clear(consumptions)
        for i = 1 to N do
            before ← checkBatteryStatus()
            execute(state)
            after ← checkBatteryStatus()
            consumptions ← consumptions ∪ {(after − before) ∗ capacity}
        end for
        avgConsumed ← average(consumptions, N)
        coefficients ← coefficients ∪ {(state, avgConsumed)}
    end for
end for
return coefficients
```

running at a certain frequency, with a known percentage of use), then the power model computes the current energy consumption as a equation of those coefficients. Those Android energy consumption models are often implemented as stand alone applications[3], which indicate the (current) energy consumption of other application running in the same device. In the next section, we present our methodology to use our models in an energy profiling tool for Android application developers.

3 Energy Consumption in Source Code

Modern programming languages offer their users powerful compilers, that included advanced optimizations, to develop efficient and fast programs. Such languages also offer advanced supporting tools, like debuggers, execution and memory profilers, so that programmers can easily detect and correct anomalies in the source code of their applications. In this section, we present one methodology that uses/adapts the (dynamic) power model defined in the previous section, to be the building block of an energy profiling tool for Android applications. The idea is to offer Android application developers an energy profiling mechanism, very much like the one offered by traditional program profilers [14]. That is to say that we wish to provide a methodology, and respective tool support, that automatically locates in the source code of the application being developed the code fragments responsible for an abnormal energy consumption.

Our methodology consists of the following steps: First, the source code of the application being monitored is instrumented with calls to the calibrated power model. Fig. 2 displays this step.

[3] Powertutor application website: `https://powertutor.org`.

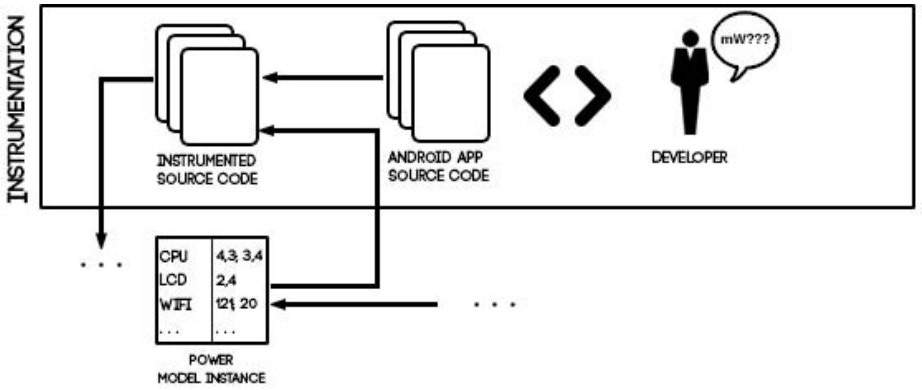


Fig. 2. The behavior of the instrumentation tool

After compiling such instrumented version of the source code, the resulting application is executed with a set of test cases. The result of such executions are statistically analyzed in order to determine which packages/methods are responsible for abnormal energy consumptions.

The source code instrumentation and execution of test cases is performed automatically as we describe in the next sections. To instrument the source code with calls to the power model, we need to model it as an API. This is discussed first.

3.1 The Model as an API

In order to be able to instrument the source code of an application, with energy profiling mechanisms, we need to adapt the current implementation of power model described in Section 2.1. That power model is implemented as a stand alone tool able to monitor executing applications. Thus, we needed to transform that implementation into an API-based software, so that its methods can be reused/called in the instrumented source code.

To adapt the power tutor implementation, we introduced a new Java class that implements the methods to be used/called by other applications and respective test cases. Those methods work as a link interface between the power consumption model and the applications source code to be monitored.

The methods implemented in the new Java class, called *Estimator*, and that are accessible to other applications are:

- traceMethod(): The implementation of the program trace .
- saveResults(): store the energy profile results in a file.
- start(): start of the energy monitoring thread.
- stop(): stop of the energy monitoring thread.

3.2 Source Code Instrumentation

Having updated the implementation of the power model so that its energy profiling methods can be called from other applications, we can now instrument an application sourse code to call them.

In order to automatically instrument the source code, we need to define the code fragments to monitor. Because we wish to do it automatically, that is by a software tool, we need to precisely define which fragments will be considered. If we consider code fragments too small, for example, a line in the source code, than, the precision of the power model may be drastically affected: a neglected amount of energy would probably be consumed. In fact, there is not a tool that we can use capable of giving power consumption estimates at a so fine grained level, with reliable results. On the other hand, we should not consider to large fragments, since this will not give a precise indication on the source code where an abnormal energy consumption exists.

We choose to monitor application methods, since they are the logical code unit used by programmers to structure the functionality of their applications. To automatize the instrumentation of the source code of an application we use the JavaParser tool[4]: it provides a simple Java front-end with tool support for parsing and abstract syntax tree construction, traversal and transformation.

We developed a simple instrumentation tool, called jInst, that instruments all methods of all Java classes of a chosen Android application project, together with the classes of an Android test project. jInst injects new code instructions, at the beginning of the method and just before a return instruction (or as the last instruction in methods with no return), as shown in the next code fragment:

```
public class Draw{
...
public void funcA(){
    Estimator.traceMethod("funcA", "Draw", Estimator.BEGIN
    );
     ...
    Estimator.traceMethod("funcA", "Draw", Estimator.END);
}
```

This code injection allows the final framework to monitor the application, keeping trace of the methods invoked and energy consumed.

3.3 Automatic Execution of the Instrumented Application

After compiling the instrumented source code an Android application is produced. When executing such application energy consumption metrics are

[4] Java parser framework webpage: `https://code.google.com/p/javaparser`.

produced. In order to automatically execute this application with different inputs, we use Android testing framework[5] that is based on jUnit.

In order to use the instrumented application and the developed *Estimator* energy class, the application needs to call methods start and stop before/after every test case is executed. Both jUnit and Android testing framework allow test developers to write a *setUp()* and a *tearDown()* methods, that are executed after a test starts and after a test ends, respectively. So, our jInst tool only needs to instrument those methods so we can measure the consumption for each test, as shown in the next example:

```
public class TestA{
...
@Override
public void setUp(){
    Estimator.start(uid);
    ...
}
...
@Override
public void tearDown(){
    Estimator.stop();
    ...
}
```

With this approach, we assure that every time a test starts, the method *Estimator.start(int uid)* is called. This method starts a thread that is going to collect information from the operating system and then apply the power consumption model to estimate the energy consumed. The *uid* argument of the method is the UID of the application in test, needed to collect the right information. The *tearDown()* is responsible for stopping the thread and saving the results.

3.4 Green-Aware Classification of Source Code Methods

Now, we need to define a metric to classify the methods according to the influence they have in the energy consumption. They are characterized as follows:

- *Green Methods:* These are the methods that have no interference in the anomalous energy consumptions. They are never invoked when the application consumes more energy than the average.
- *Red Methods:* Every time they are invoked, the application has anomalous energy consumption. They can be invoked when the application has bellow the average energy consumption as well, but no more than 30% of the times. They are supposed to be the methods with bigger influence in the anomalous energy consumption.

[5] Android testing web page: `https://developer.android.com/tools/testing/index.html`.

– *Yellow Methods:* The methods that are invoked in other situations: mostly invoked when the application power consumption is bellow the average.

This representation of methods is also extensible for classes, packages and projects. In other words, the classification of classes depends on the set of methods they have, and so packages depend on their respective classes, as the projects are classified according to their packages. If a class has more than 50% of its methods classified as Red Methods, then it is a red class. With more 50% of them as green, it is a Green Class. Otherwise, it is a Yellow class. Packages and projects follow the same approach.

4 GreenDroid: An Android Framework for Energy Profiling

At this point, we have power source code instrumentation, consumption measurement, method tracing and automatic test execution using the Android testing framework. The final result should be a tool that works automatically, and does all this tasks incrementally. After that, it retrieves the information previously saved and shows the results obtained. This section explains the workflow of the framework, along with the information obtained at every point, and how the results are obtained and generated.

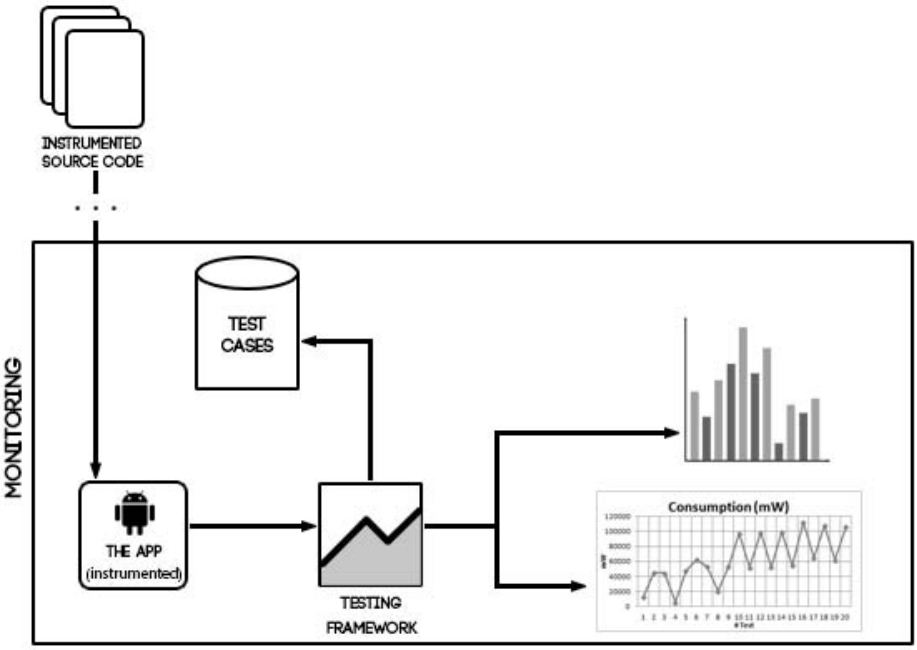


Fig. 3. The behavior of the monitoring framework

4.1 Workflow

After the source code of the application and the tests are instrumented as described in section 2, the framework executes a set of sequential steps to show the conclusive results, that being:

- *Execute the tests*: This is the starting point for the framework. The tests will be executed twice, the first time to get the trace (list of invoked methods) and the second to measure power consumption, so that the tracing overhead does not affect measuring. The results will be saved in files (one for each test), containing a list of the methods invoked, along with the number of times it was invoked, and also the execution time of the test and the energy consumed, in mW.
- *Merge the results*: After all the tests executed (twice), the framework would have generated a set of files as big as the number of tests. For convenience, the tests will be merged in one file to be read, parsed and the information extracted once.
- *Classify the methods*: At this point, the framework will get the values read from the file and classify them (and respective classes, packages and projects) according to the categories described in section 3.4.
- *Generate the results*: The framework will than generate a graphical representation of the source code components, giving them different colors according to its green-aware classification.

This steps are all represented in Figure 3, that represents how the application, after instrumented, generates the results for measuring and tracing of the test cases defined with the Android testing framework.

5 Results

This section shows the results of running multiples tests with our framework from an open source Android application called *0xBenchmark*[6]. This application allowed us to simulate different kinds of executions. We managed to run 20 different tests, and each test had a different set of methods invoked (execution trace). So, for each test we managed to keep the trace, the power consumption, the time a test executed and the number of times a method was invoked. It is important to mention that the values presented in the charts reflect, for each test case, an average of several measures. It makes sense to do it since this is a statistical approach. If we look at the Fig. 4, we can see that different tests have different values of power consumption. One could think that the tests with bigger values for power consumption are the ones with bigger execution times, and so the energy consumed per unit of time would be nearly the same for all the tests, but Fig. 5 shows that the consumption per second varies between the tests.

[6] 0xbench can be found at `http://0xbenchmark.appspot.com`.

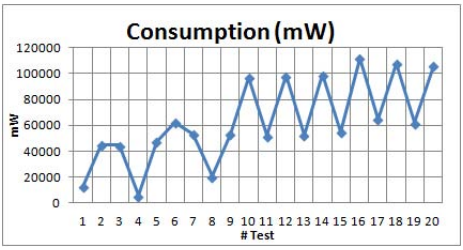


Fig. 4. Total consumption per test

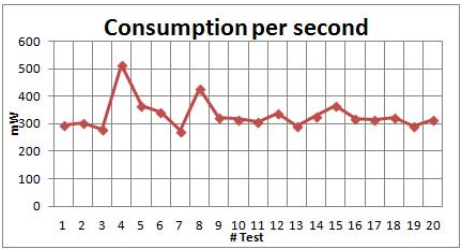


Fig. 5. Consumption per second

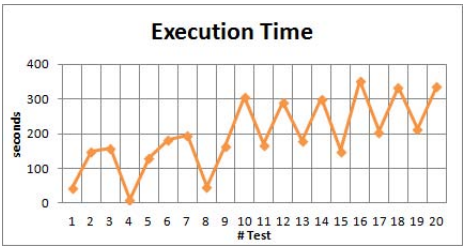


Fig. 6. Execution time

So, these are very good indicators, they allow us to conclude that execution time has an influence in the power consumption, but it is not the only relevant factor. In fact, it might not be one of the most relevant.

So, with the approach described in Section 3 to detect tests with excessive power consumption, we can get a percentage for each method that reflects its influence in energy anomalous tests. The framework will then assign that percentage to the respective method, taking into account the test results, and display a sunburst diagram like the one in Fig. 7 (similar to the approach presented in MZoltar [15]) that allows the developer to quickly identify the most energy anomalous methods.

6 Related Work

In the past years the investigation of power consumption in *smartphones* has been increasing. Several research works indicate that power consumption modeling and energy-aware software are getting their importance in the investigation scope. It is possible to find different tools designed to estimate the required energy for an application to do its tasks. The majority of them focus on the Android based *smartphones*, mostly because it is an open source OS[7] and statistics reveal that the percentage of selling is much higher for Android devices than any

[7] An Android overview can be found at `http://www.openhandsetalliance.com/android_overview.html`.

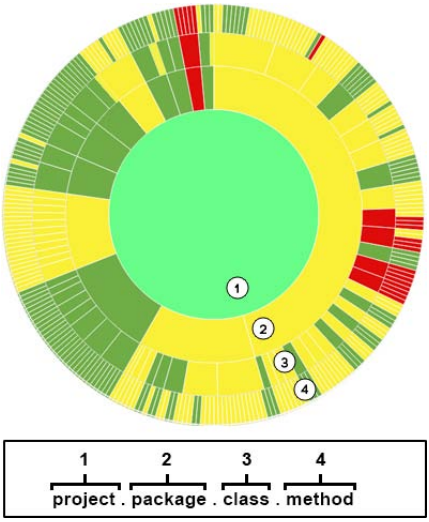


Fig. 7. Sunburst diagram (and how to interpret it)

other[8]. In fact, in the second quarter of 2013 almost 80% of the market share belonged to Android devices.

Most of the research works start by identifying the hardware components of the device with significant influence on its energy consumption. We have the example of Power Tutor [9], that is the starting point for many other research works, but also DevScope [13] and related tools (AppScope [12] and UserScope [16]). These tools have a power consumption model relating the different hardware components of a device to its different states and consequent power consumption values. The main difference lies in the implementation of the model: one works as an independent Android application, the other is a Linux kernel module, but both of them focus on collecting the hardware components' usage information from the file system.

There are other examples of works based on power consumption models and its applications in different areas ([11, 17–19]), however none of them is as powerful as the remaining ones. Another interesting example, SEMO [20], has a similar behavior to Power Tutor, but does not use power consumption models. Instead, it is focused in battery discharge level, and its results are less reliable.

Other works [21, 22] demonstrate that it is possible to have different values on energy consumption for different softwares designed to do the same tasks. So this can be a very good indicator that helping developers choose the most energy-aware solution for a software implementation is of great importance. In fact, this has been demonstrated in [7].

[8] Information about global smartphone shipments can be found at `http://techcrunch.com/2013/08/07/android-nears-80-market-share-in-global-smartphone-shipments-as-ios-and-blackberry-share-slides-per-idc`.

7 Conclusions and Future Work

The energy consumption is nowadays of paramount importance. This is also valid to software, and specially for applications running in mobile devices. Indeed the most used platform is clearly the Android and thus we have devote our attention to its applications.

Given the innumerable quantity of Android versions and devices, our approach is to create a dynamic model that can be used in any device and any Android system, and that can give information to the developers about the methods he/she is writing that consume the most energy. We have created a tool that can automatically calibrate the model for every phone, and another to automatically annotate any application source code so the programmer can have energy consumption measures with almost no effort.

With this work we were able to show that the execution time is highly correlated to the total energy consumption of an application. Although this seems obvious, until now it was only speculative. We have also shown that the total time and energy the application takes to execute a set of tasks does not indicate the worst methods. To find them, it is necessary to apply the techniques we now propose, measuring this consumption by second and computing the worst methods, called red methods.

Nevertheless, there is still work to be done. Indeed it is still necessary to evaluate the precision of the results of our consumption measurements. Since we do not use real measurements from the physical device components, we still need to confirm that the results we can compute are accurate enough.

References

1. Jones, N.D.: An introduction to partial evaluation. ACM Comput. Surv. 28(3), 480–503 (1996)
2. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. ACM Trans. Program. Lang. Syst. 28(6), 990–1034 (2006)
3. Krall, A.: Efficient javavm just-in-time compilation. In: International Conference on Parallel Architectures and Compilation Techniques, pp. 205–212 (1998)
4. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theoretical Computer Science 73, 231–248 (1990)
5. Saraiva, J., Swierstra, D.: Data Structure Free Compilation. In: Jähnichen, S. (ed.) CC 1999. LNCS, vol. 1575, pp. 1–17. Springer, Heidelberg (1999)
6. Fernandes, J.P., Saraiva, J., Seidel, D., Voigtländer, J.: Strictification of circular programs. In: Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, pp. 131–140. ACM (2011)
7. Pinto, G., Castor, F., Liu, Y.D.: Mining questions about software energy consumption. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp. 22–31. ACM, New York (2014)
8. Carção, T.: Measuring and visualizing energy consumption within software code. In: Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC. IEEE (to appear, 2014)

9. Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R.P., Mao, Z.M., Yang, L.: Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In: Proc. Int. Conf. Hardware/Software Codesign and System Synthesis (October 2010)
10. Dong, M., Zhong, L.: Self-Constructive High-Rate System Energy Modeling for Battery-Powered Mobile Systems. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys 2011 (2011)
11. Kjaergaard, M.B., Blunck, H.: Unsupervised Power Profiling for Mobile Devices. In: 8th International ICST Conference, Copenhagen, Denmark (December 2011)
12. Yoon, C., Kim, D., Jung, W., Kang, C., Cha, H.: AppScope: Application Energy Metering Framework for Android Smartphones using Kernel Activity Monitoring. In: USENIX Annual Technical Conference (USENIX ATC 2012) (June 2012)
13. Jung, W., Kang, C., Yoon, C., Kim, D., Cha, H.: DevScope: A Nonintrusive and Online Power Analysis Tool for Smartphone Hardware Components. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2012) (October 2012)
14. Runciman, C., Röjemo, N.: Heap Profiling for Space Efficiency. In: Launchbury, J., Sheard, T., Meijer, E. (eds.) AFP 1996. LNCS, vol. 1129, pp. 159–183. Springer, Heidelberg (1996)
15. Machado, P., Campos, J., Abreu, R.: MZoltar: Automatic Debugging of Android Applications. In: First International Workshop on Software Development Lifecycle for Mobile (DeMobile), Co-located with European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Saint Petersburg, Russia (2013)
16. Jung, W., Kim, K., Cha, H.: UserScope: A Fine-grained Framework for Collecting Energy-related Smartphone User Contexts. In: IEEE International Conference on Parallel and Distributed Systems (ICPADS 2013) (December 2013)
17. Kim, D., Jung, W., Cha, H.: Runtime Power Estimation of Mobile AMOLED Displays. In: Design, Automation & Test Europe (DATE 2013) (March 2013)
18. Carroll, A., Heiser, G.: An Analysis of Power Consumption in a Smartphone. In: USENIXATC 2010 Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (2010)
19. Zhang, L., Gordon, M.S., Dick, R.P., Mao, Z.M., Dinda, P., Yang, L.: ADEL: An Automatic Detector of Energy Leaks for Smartphone Applications. In: IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (2012)
20. Ding, F., Xia, F., Zhang, W., Zhao, X., Ma, C.: Monitoring Energy Consumption of Smartphones. In: Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing (October 2012)
21. Corral, L., Georgiev, A.B., Sillitti, A., Succi, G.: Method Reallocation to Reduce Energy Consumption: An implementation in Android OS. In: Symposium on Applied Computing 2014 (2014)
22. Noureddine, A., Rouvoy, R., Seinturier, L.: Unit Testing of Energy Consumption of Software Libraries. In: Symposium on Applied Computing 2014 (2014)

Effect Capabilities for Haskell

Ismael Figueroa[1,2,⋆], Nicolas Tabareau[2], and Éric Tanter[1,⋆⋆]

[1] PLEIAD Laboratory, Computer Science Department, University of Chile, Santiago, Chile
[2] ASCOLA Group, Inria, Nantes, France

Abstract. Computational effects complicate the tasks of reasoning about and maintaining software, due to the many kinds of interferences that can occur. While different proposals have been formulated to alleviate the fragility and burden of dealing with specific effects, such as state or exceptions, there is no prevalent robust mechanism that addresses the general interference issue. Building upon the idea of capability-based security, we propose effect capabilities as an effective and flexible manner to control monadic effects and their interferences. Capabilities can be selectively shared between modules to establish secure effect-centric coordination. We further refine capabilities with type-based permission lattices to allow fine-grained decomposition of authority. We provide an implementation of effect capabilities in Haskell, using type classes to establish a way to statically share capabilities between modules, as well as to check proper access permissions to effects at compile time. We exemplify how to tame effect interferences using effect capabilities, by treating state and exceptions.

1 Introduction

Computational effects (*e.g.* state, I/O, and exceptions) complicate reasoning about, maintaining, and evolving software. Even though imperative languages embrace side effects, they generally provide linguistic means to control the potential for effect interference by enforcing some forms of encapsulation. For instance, the private attributes of a mutable object are only accessible to the object itself or its closely-related peers. Similarly, the stack discipline of exception handling makes it possible for a procedure to hide exceptions raised by internal computation, and thereby protect it from unwanted interference from parties that are not directly involved in the computation.

We observe that all these approaches are *hierarchical*, using module/package nesting, class/object nesting, inheritance, or the call stack as the basis for confining the overall scope of effects. This hierarchical discipline is sometimes inappropriate, either too loose or too rigid. Consequently, a number of mechanisms that make it possible to either cut across or refine hierarchical boundaries have been devised. A typical example mechanism for *loosening* the hierarchical constraints is friendship declarations in C++. Exception handling in Standard ML—with the use of dynamic classification [7] to prevent unintended access to exception values—is an example of a mechanism that *strengthens* the protection offered by the hierarchical stack discipline.

⋆ Funded by a CONICYT-Chile Doctoral Scholarship.
⋆⋆ Partially funded by FONDECYT project 1110051.

F.M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 92–108, 2014.

Exploiting the intuitive affinity between encapsulation mechanisms and access control security, we can see classical approaches to side effect encapsulation as corresponding to *hierarchical protection domains*. The effective alternative in the security community to transcend hierarchical barriers is *capability-based security*, in which authority is granted selectively by communicating unforgeable tokens named capabilities [11,13]. Seen in this light, the destructor of an exception value type in Standard ML is a capability that grants authority to inspect the internals of values of this type [6]. The destructor, as a first-class value itself, can be flexibly passed around to the intended parties. Friendship declarations in C++ can also be seen as a static capability-passing mechanism.

Following this intuition we propose *effect capabilities*, in the context of Haskell[1], for flexibly and securely handling computational effects. Effect capabilities are first-class unforgeable values that can be passed around in order to establish secure effect-related interaction channels. The prime focus of effect capabilities is to guarantee, through the type system, that there is no unauthorized access to a given effectful operation. Authorization is initially granted through static channel sharing at the module level, allowing detection of violations at compile time. We do not focus on dynamic sharing of capabilities, because this can only be done by modules that were already trusted at compile time.

We start illustrating the main problem addressed by effect capabilities in Haskell: the issue of *effect interference* in the monad stack (Section 2). Then we present the main technical development: a generic framework for capabilities and permissions, which can be statically shared between modules (Section 3). In this framework we combine several existing techniques, along with two novel technical contributions. First, a user-definable lattice-based permission mechanism that checks access at compile time using type class resolution (Section 3.2). And second, a static secret sharing mechanism implemented using type classes and mutually recursive modules (Section 3.3). Finally, effect capabilities are implemented using this framework in the particular case of monadic operations (Section 4), and we illustrate how to implement private and shared state (Section 4.1 and Section 4.2) as well as protected exceptions (Section 4.3).

2 Effect Interference in Monadic Programming

In this section we illustrate the problem of effect interference in monadic programming. We start with a brief description of monadic programming in Haskell (Section 2.1). Then we illustrate the particular issue of state interference (Section 2.2), also showing that the currently accepted workaround is not scalable (Section 2.3). Finally, we illustrate the issue of exception interference (Section 2.4).

2.1 Monadic Programming in a Nutshell

Monads [15,25] are the mechanism of choice to embed and reason about computational effects such as state, I/O or exception handling, in purely functional languages like

[1] Implementation on the GHC compiler available online at: http://pleiad.cl/effectcaps

Haskell. Using monad transformers [12] it is possible to modularly create a monad that combines several effects. A monad transformer is a type constructor used to create a *monad stack* where each layer represents an effect. Monadic programming in Haskell revolves around the standard MTL library, which provides a set of monad transformers that can flexibly be composed together. Typically a monad stack has either the $Identity$, or the IO monad at its bottom. When using monad transformers it is necessary to establish a mechanism to access the effects of each layer. We now briefly describe current mechanisms; for a detailed description see [21].

Explicit Lifting. A monad transformer t must define the $lift$ operation, which takes a computation from the underlying monad m, with type $m\ a$, into a computation in the transformed monad, with type $t\ m\ a$. Explicit uses of $lift$ directly determine which layer of the stack is being used.

Implicit Lifting. To avoid explicit uses of $lift$, one can associate a type class with each particular effect, defining a public interface for effect-related operations. Using the type class resolution mechanism, the monadic operations are routed to the first layer of the monad stack that satisfies a given class constraint. This is the mechanism used in the transformers from MTL, where the implicit liftings between them are predefined.

Tagged Monads. In this mechanism the layers of the monad stack are marked using type-level tags. The tags are used to improve implicit lifting, in order to route operations to specifically-tagged layers, rather than the first layer that satisfies a constraint [16,23,21]. In this work we focus only on the standard lifting mechanisms, which underlie the implementations of tagged monads, and leave for future work the integration of type-level tags and effect capabilities.

2.2 State Interference

As a running example to illustrate the issue of effect interference as well as its solution using effect capabilities, we consider the implementation of two monadic abstract data types (ADTs). These are a queue of integer values, with operations $enqueue$ and $dequeue$; and a stack, also of integer values, with operations $push$ and pop.

Regarding state, ideally each ADT should have a private state that cannot be modified by components external to the module. Before we describe the implementation, let us recall the standard state transformer and its associated type class:

$$\begin{array}{l}\textbf{newtype}\ StateT\ s\ m\ a = StateT\ (s \to m\ (a, s)) \\ \textbf{class}\ Monad\ m \Rightarrow MonadState\ s\ m \mid m \to s\ \textbf{where} \\ \quad get :: m\ s \\ \quad put :: s \to m\ () \end{array}$$

A typical and reusable implementation of these ADTs is defined using implicit lifting. A straightforward implementation of the structures' operations is as follows:

$$
\begin{aligned}
&enqueue_1 :: MonadState\ [Int]\ m \Rightarrow Int \to m\ ()\\
&enqueue_1\ n = \mathbf{do}\ \{\, queue \leftarrow get; put\ \$\ queue +\!\!+ [n]\,\}\\
&dequeue_1 :: MonadState\ [Int]\ m \Rightarrow m\ Int\\
&dequeue_1\ \ \ = \mathbf{do}\ \{\, queue \leftarrow get; put\ \$\ tail\ queue; return\ \$\ head\ queue\,\}\\
&push_1 :: MonadState\ [Int]\ m \Rightarrow Int \to m\ ()\\
&push_1\ n = \mathbf{do}\ \{\, stack \leftarrow get; put\ (n : stack)\,\}\\
&pop_1 :: MonadState\ [Int]\ m \Rightarrow m\ Int\\
&pop_1\ \ \ \ = \mathbf{do}\ \{\, stack \leftarrow get; put\ \$\ tail\ stack; return\ \$\ head\ stack\,\}
\end{aligned}
$$

Thanks to implicit lifting, the functions can be evaluated in any monad stack m that fulfills the $MonadState\ [Int]$ constraint. With the intent of giving each ADT its own private state, we define a monad stack M with two state layers.

$$\mathbf{type}\ M = StateT\ [Int]\ (StateT\ [Int]\ Identity)$$

However, using both ADTs in the same program leads to state interference. The problem is that implicit lifting will route both $enqueue$ and $push$ operations to the first layer of M. For example, evaluating:

$$
\begin{aligned}
client_1 = \mathbf{do}\ & push_1\ 1\\
& enqueue_1\ 2 \quad \text{-- value is put into the state layer used by the stack}\\
& x \leftarrow pop_1\\
& y \leftarrow pop_1 \quad \text{-- should raise error because stack should be empty}\\
& return\ (x + y)
\end{aligned}
$$

yields 3 instead of throwing an error when attempting to pop_1 the empty stack. To address this issue, one of the ADTs must use explicit lifting to use the second state layer, for instance we can modify the queue operations:

$$
\begin{aligned}
&enqueue'_1\ n = \mathbf{do}\ \{\, queue \leftarrow lift\ get; (lift \circ put)\ (queue +\!\!+ [n])\,\}\\
&dequeue'_1\ n = \mathbf{do}\ \{\, queue \leftarrow lift\ get; (lift \circ put)\ (tail\ queue); return\ (head\ queue)\,\}
\end{aligned}
$$

However, as discussed by Schrijvers and Oliveira [21], this solution is still unsatisfactory. First, the approach is *fragile* because the number of $lift$ operations is tightly coupled to the particular monad stack used, thus hampering modularity and reusability. And second, because the monad stack is *transparent*, meaning that nothing prevents $enqueue'_1$ or $dequeue'_1$ to use get and put operations that are performed on the first state layer. Conversely, nothing prevents $push_1$ or pop_1 from accessing the second state layer. In fact, any monadic component can modify the internal state of these structures.

2.3 State Encapsulation Pattern

To the best of our knowledge, the current practice to implement private state in Haskell—in order to avoid issues like the one above—is to define a custom state-like monad transformer and hide its data constructor. For instance, a polymorphic queue ADT can be implemented based on a new $QueueT$ monad transformer, which reuses the implementation of $StateT$ to represent the queue as a list of values:[2]

[2] We do not show it here, but we use the $GeneralizedNewtypeDeriving$ extension of GHC to derive the necessary instances of the $Monad$ and $MonadTrans$ type classes.

```
newtype QueueT s m a = QueueT (StateT [s] m a) deriving ...
```

The definitions of *enqueue* and *dequeue* are similar to those already presented, but let us consider their types:

```
enqueue_2 :: s → QueueT s m ()
dequeue_2 :: QueueT s m s
```

Because these definitions are tied specifically to a monad stack where $QueueT$ (resp. $StackT$) is on top, another requirement to integrate with implicit lifting is to declare a new type class $MonadQueue$ (resp. $MonadStack$), whose canonical instance is given by $QueueT$ (resp. $StackT$).

In short, a $Queue$ module that encapsulates its state can be defined as:

```
module Queue (QueueT (), MonadQueue (..), enqueue, dequeue) where
newtype QueueT s m a = QueueT (StateT [s] m a) deriving ...
class Monad m ⇒ MonadQueue s m where
  enq :: s → m ()
  deq :: m s
instance Monad m ⇒ MonadQueue s (QueueT s m) where
  enq s = QueueT $ StateT $ λq → return ((), q ++ [s])
  deq   = QueueT $ StateT $ λq → return (head q, tail q)
enqueue :: (MonadQueue [Int] m) ⇒ Int → m ()
enqueue = enq
dequeue :: (MonadQueue [Int] m) ⇒ m Int
dequeue = deq
```

Declaring $QueueT$ as instance of $MonadQueue$ requires the implementation of enq and deq. As $QueueT$ relies on the standard state transformer $StateT$, the implementation is straightforward. The crucial point to ensure proper encapsulation is that *the module does not export the $QueueT$ data constructor*. This is explicit in the module signature as $QueueT$ (), which means that only the type $QueueT$ is exported, but its data constructors remain private.

Avoiding interference. Using the $QueueT$ and $StackT$ transformers, as well as the $MonadQueue$ and $MonadStack$ type classes defined using this pattern,we can rephrase our previous example in order to avoid state interference:

```
import Queue
type M = QueueT Int (StackT Int Identity)
client_2 = do push 1
              enqueue 2
              x ← pop
              y ← pop    -- error: popping from empty stack
              return (x + y)
```

Scalability Issues. The main issue of the state encapsulation pattern is that it is not scalable. To properly integrate $MonadQueue$ and $MonadStack$ with implicit lifting, we

would need to declare $QueueT$ and $StackT$ as an instance of every other effect-related type class, and to make every other monad transformer an instance of $MonadQueue$ and $MonadStack$ as well. If we consider only the 7 standard transformers in the MTL, this effort amounts to 28 instance declarations! (14 instances for each encapsulated state) Moreover, when using non-standard transformers, it may not be possible to anticipate all the required combinations; therefore the burden lies on the user of such libraries to fill in the gaps. We are not the first to note the quadratic growth of instance declarations with this approach, *e.g.* Hughes dismisses monads as an option to implement global variables in Haskell for this very reason [8].

2.4 Exception Interference

Another form of effect interference can occur in a program that uses exceptions and exception handlers. The problem is that due to the dynamic nature of exceptions and handlers, it is possible for exceptions to be inadvertently caught by unintended handlers—for instance, by "catch-all" handlers. As an illustration, consider an application where the queue is used by a $consume$ function.

$$
\begin{aligned}
&consume :: (MonadQueue\ Int\ m, MonadError\ String\ m) \Rightarrow m\ Int \\
&consume = \mathbf{do}\ x \leftarrow dequeue \\
&\qquad\qquad \mathbf{if}\ (x < 0)\ \mathbf{then}\ throwError\ \text{“}Process\ error\text{”} \\
&\qquad\qquad\qquad\qquad \mathbf{else}\ \ return\ x
\end{aligned}
$$

This function checks an invariant that values should be positive, and throws an exception otherwise. Further assume that another $process$ function relies on $consume$.

$$
\begin{aligned}
&process :: (MonadQueue\ Int\ m, MonadError\ String\ m) \Rightarrow Int \rightarrow m\ Int \\
&process\ val = consume\ `catchError`\ (\lambda e \rightarrow return\ val)
\end{aligned}
$$

Here $process$ uses an exception handler, $catchError$, to get a default value val whenever $consume$'s invariant does not hold. Consider now a variant of $dequeue$ that raises an exception when trying to retrieve a value from an empty queue. Its type is

$$dequeue :: (MonadQueue\ s\ m, MonadError\ String\ m) \Rightarrow m\ s$$

In this scenario, exception interference will occur because the same exception effect is used to signal two different issues. Consider the following program:

$$
\begin{aligned}
program_1 = \mathbf{do}\ &enqueue\ (-10) \\
&process\ 23
\end{aligned}
$$

When evaluated, $program_1$ yields 23 because the value in the queue breaks the invariant of $consume$, triggering the handler of $process$. Now, consider a second program:

$$program_2 = process\ 23$$

which will also yield 23, but because the queue was empty—not because the invariant of $consume$ was broken. In this setting, it is not possible to assert non-emptiness of the queue, because exceptions get "swallowed" by another handler. Similar to state interference, current solutions rely on custom exception transformers and explicit lifting.

As argued by Harper [6], the standard semantics of exceptions difficult the modular composition of programs because of the potentially modified exception flows. Indeed, issues like this has been identified in the context of aspect-oriented programming [3]. In Section 4.3 we show how effect capabilities allows us to define exceptions that, like in Standard ML, can be protected from unwanted interception.

3 A Generic Static Framework for Capabilities and Permissions

This section presents the main technical development of this work: a generic framework for capabilities, upon which effect capabilities are built, in the next section. First, we define capability-based access as a computational effect (Section 3.1). Then, we refine simple capabilities with type-based and user-definable permission lattices (Section 3.2); and show how capabilities can be shared between modules (Section 3.3). Finally we describe how the framework supports two key features of capabilities-based mechanisms: delegation and attenuability (Section 3.4).

3.1 Private Capabilities as a Computational Effect

A private capability is a singleton type whose type is public but whose constructor is private. For instance, consider the capability for read/write access to some state:

$$\mathbf{data}\ RWCap = RWCap$$

We turn this capability into a notion of *protected computations* by using a specific reader monad transformer for capabilities, $CapT$. Using the reader transformer allows us to embed the actual capability used to run a computation into the read-only environment bound to a reader monad. Similar to state encapsulation, $CapT$ is defined in terms of the canonical reader monad transformer $ReaderT$.

$$\mathbf{newtype}\ CapT\ c\ m\ a = CapT\ (ReaderT\ c\ m\ a)\ \mathbf{deriving}\ ...$$
$$fromCapT :: c \to CapT\ c\ m\ a \to m\ a$$
$$fromCapT\ !\ c\ (CapT\ ma) = runReaderT\ ma\ c$$

A capability has a public type but a private value, but as Haskell is lazy, a malicious module can always forge a capability for which it has no access by passing $\perp$ as the capability argument to evaluate $fromCapT$.[3] To avoid this situation, we use a strictness annotation ! in the implementation of $fromCapT$. Note this issue would not be present in a strict setting. As an example, consider a module A that uses $RWCap$ to restrict access to a state monad holding a value of type s.

$$\mathbf{module}\ A\ (getp, putp, RWCap\ ())\ \mathbf{where}$$
$$\mathbf{data}\ RWCap = RWCap$$
$$getp :: CapT\ RWCap\ (State\ s)\ s$$
$$putp :: s \to CapT\ RWCap\ (State\ s)\ ()$$

A module B that imports A will get access to both operations, but will not be able to perform any of them because it will lack the $RWCap$ value, which can only be constructed in the context of module A.

[3] $\perp$ is an expression that pertains to all types, and directly fails with an error if evaluated. Hence, it can pass as any capability, fulfilling the expected type of $fromCapT$.

3.2 Private Lattice of Permissions

Capabilities are unforgeable authority tokens that unlock specific monadic operations. Ideally, a system should follow the *principle of least privilege* [18], which in our context means that it should not be necessary to have write permissions just to read the value of a state monad; and conversely, reading access is not necessary to update such a state.

We now refine the model of capabilities with the possibility to attach *permissions* to a capability, in order to allow a finer-grained decomposition of authority. A permission denotes the subset of operations that the capability permits. Now capabilities are defined as type constructors with a single argument, the permission; and permissions are defined as singleton types.

Permission Lattices. Permissions can be organized in a lattice specified by a $\sqsupset$ type class. $\sqsupset$ is a simple reflexive and transitive relation on types defined as:[4]

```
class    a ⊐ b
instance a ⊐ a   -- generic instance for reflexivity
```

The type class $\sqsupset$ must not be public because it would allow a malicious user to add a new undesirable relation in the lattice to effectively bypass permission checking altogether. Still, we want to be able to impose constraints based on the private lattice in other modules. To do that, we define a public lattice $\supset$ that exports the private lattice without being updatable from the outside of the module, because extending it always requires to define an instance on the private lattice.[5]

```
class    a ⊐ b ⇒ a ⊃ b
instance a ⊐ b ⇒ a ⊃ b
```

Permission Lattices in Practice. Going back to the previous example, we now define the $RWCap$ capability, as well as the $ReadPerm$, $WritePerm$ and $RWPerm$ permissions, denoting read-only, write-only and read-write access, respectively. We also define the private and public permission lattices $\sqsupset_{RW}$ and $\supset_{RW}$, for state access permissions:

```
module RWLattice (⊃_RW, RWCap (), ReadPerm, WritePerm, RWPerm) where

data RWCap p   = RWCap p
data ReadPerm  = ReadPerm
data WritePerm = WritePerm
data RWPerm    = RWPerm

  -- private lattice
class    a ⊐_RW b
instance a ⊐_RW b

  -- private instances, not updateable externally
instance RWPerm ⊐_RW ReadPerm
instance RWPerm ⊐_RW WritePerm

  -- public lattice
class    a ⊐_RW b ⇒ a ⊃_RW b
instance a ⊐_RW b ⇒ a ⊃_RW b
```

[4] Unlike logic programming, transitivity cannot be deduced from a generic instance, due to an ambiguity issue during type class resolution; hence all pairs of the relation must be explicit.

[5] Haskell type classes are *open*, that is, instances of publicly exported type classes can be added in any part of the system. Private type classes are confined to the module that defines them.

Crucially, we use module encapsulation to hide the private lattice $\sqsupset_{RW}$. Thus we only export the public lattice $\supset_{RW}$, which can be used as a regular class constraint. Otherwise, obtaining write-access from a read-only permission is as simple as:

$$\begin{array}{l}\textbf{module } \mathit{BypassRW} \textbf{ where}\\ \textbf{import } \mathit{RWLattice}\\ \textbf{instance } \mathit{ReadPerm} \sqsupset_{RW} \mathit{WritePerm}\end{array}$$

Using the public permission lattice allows developers to impose fine-grained access constraints using the public type class $\supset_{RW}$. For instance, the functions $getp$ and $putp$ can be refined as:

$$\begin{array}{l}\mathit{getp} :: \mathit{perm} \supset_{RW} \mathit{ReadPerm} \Rightarrow \mathit{CapT}\ (\mathit{RWCap}\ \mathit{perm})\ (\mathit{State}\ s)\ s\\ \mathit{putp} :: \mathit{perm} \supset_{RW} \mathit{WritePerm} \Rightarrow s \to \mathit{CapT}\ (\mathit{RWCap}\ \mathit{perm})\ (\mathit{State}\ s)\ ()\end{array}$$

As a final remark, recall from Section 3.1 that type class resolution statically checks for proper permissions when a computation is evaluated using $\mathit{fromCapT}$.

Capabilities as namespaces for permissions. Capability constructors, such as $RWCap$, may appear superfluous, because we are interested in the permissions for protected operations. However, such constructors serve the crucial role of serving as namespaces for permissions. This allows a module to have restricted read-only access to some state, while still having full read-write access to another state.

3.3 Static Sharing of Capabilities

We now describe how to go beyond private capabilities and support the ability to allow specific modules to have access to capabilities. The issue addressed here is that most module systems, including that of Haskell, do not make it possible to expose bindings to explicitly-designated modules. For example, as we illustrate in Section 4.2, for efficiency reasons a $Queue$ module can provide read-only access to its internal state to a $PriorityQueue$ module, which simply acts as another interface on top of the queue.

Conceptually, the idea of static sharing is to use public accessors to selectively share capabilities. However this requires a trusted mechanism by which modules can be identified properly by the accessors. The development of this idea yields a mechanism for *static* message passing, using type classes, loosely inspired by the π-calculus notion of messages and channels [19].

Message sending as type class instances. In analogy with capabilities, a channel is just a singleton type whose type is public, but whose (unique) value is private. Channels are governed by the $Channel$ monad reader which prevents from the use of $\bot$:

$$\begin{array}{l}\textbf{newtype } \mathit{Channel}\ \mathit{ch}\ a = \mathit{Channel}\ (\mathit{Reader}\ \mathit{ch}\ a) \textbf{ deriving } ...\\ \mathit{fromChannel} :: \mathit{ch} \to \mathit{Channel}\ \mathit{ch}\ a \to a\\ \mathit{fromChannel}\ !\ \mathit{ch}\ (\mathit{Channel}\ \mathit{ma}) = \mathit{runReader}\ \mathit{ma}\ \mathit{ch}\end{array}$$

We define a type class $Send$ for message sending:

$$\begin{array}{l}\textbf{class } \mathit{Send}\ \mathit{ch}\ c\ p \textbf{ where}\\ \quad \mathit{receive} :: p \to \mathit{Channel}\ \mathit{ch}\ (c\ p)\end{array}$$

This type class requires three types: a channel ch, a capability c, and a permission p; and it provides the $receive$ method. Sending a message of type $c\ p$ on ch amounts to declaring an instance $Send\ ch\ c\ p$. Conversely, receiving a message of type $c\ p$ on ch amounts to applying the function $receive$ to p and getting the value back using $fromChannel$.[6] Observe that the messaging protocol is rather asymmetric, because capabilities are sent statically by declaring type classes instances, but are received dynamically by calling $receive$. This is not problematic because type class resolution will check that all calls to $receive$ are backed up by an instance of $Send$, or else type checking will fail. Therefore, the protocol ensures that one module can only receive a message that has been sent to it. [7]

For instance, following the motivation example, the $Queue$ module can send the $RWCap$ capability with read permission to the $PQChan$ channel provided by the $PriorityQueue$ module (full example in Section 4.2):

$$\begin{aligned}&\textbf{instance}\ Send\ PQChan\ RWCap\ ReadPerm\\ &\quad\textbf{where}\ receive\ ReadPerm = return\ \$\ RWCap\ ReadPerm\end{aligned}$$

Of course, this mechanism is less expressive than message passing in process calculi—only one message of type $c\ p$ can be sent on a specific channel—but it is sufficient for our purposes since permissions are singleton types.

3.4 Delegation and Attenuability

Capabilities-based mechanisms feature two characteristics called *delegation* and *attenuability* [11]. In combination, these characteristics allow an entity to transmit (a restricted version of) its capabilities to another entity in the system. We describe how these characteristics are supported in the framework.

Delegation. The sharing mechanism allows for static delegation of capabilities. A module B that receives a capability from other module A, can in turn transmit the capability to another module C. This is sound because B cannot transmit more capabilities than those it receives from A. Figure 1 shows static delegation of the $RWCap\ ReadPerm$ capability.

Attenuability. A capability with a high permission in a permission lattice can be attenuated into another capability with a lower permission implied by the former. To support attenuability, we force capabilities to define a function $attenuate$ using the type class:

$$\begin{aligned}&\textbf{class}\ Capability\ c\ \supset \mid c \to \supset\ \textbf{where}\\ &\quad attenuate :: p_1 \supset p_2 \Rightarrow c\ p_1 \to p_2 \to c\ p_2\end{aligned}$$

Here $attenuate$ degrades the permission if it respects the lattice structure of $\supset$. If module A needs to provide a limited version of a capability to module B it can provide a

[6] The expected result type $c\ p$ has to be provided explicitly, because messages of different types can be sent on the same channel.

[7] We rely on GHC support for mutually recursive modules for inter-module communication. See http://www.haskell.org/ghc/docs/latest/html/users_guide/separate-compilation.html

```
module A where
import B   -- to send capability to BChannel
instance Send BChannel RWCap ReadPerm where
  receive ReadPerm = return $ RWCap ReadPerm
module B where
import A   -- to get the capability sent by A
import C   -- to send capability to CChannel
data BChannel = BChannel
instance Send CChannel RWCap ReadPerm where
  receive ReadPerm = return $ (fromChannel BChannel $ receive ReadPerm)
module C where
import B   -- to get the capability sent by B
data CChannel = CChannel
cap :: RWCap ReadPerm
cap = fromChannel CChannel $ receive ReadPerm
```

Fig. 1. Static delegation of capabilities

sub-permission based on the existing permission lattice using the function $attenuate$. Note that $\supset$ is a parameter of the class because different capabilities may be defined on different lattices, but the functional dependency $c \rightarrow \supset$ imposes that only one lattice is attached to a capability. If the required permission is not already provided by the existing lattice, one can always define a refined lattice (and redefine associated functions).

4 Effect Capabilities: Upgrading Monads with Capabilities

We now delve into the main subject of this work: how to use capabilities to control monadic effects and their interferences in an effective and flexible manner. Building upon the generic capabilities framework, which can be used to restrict access to arbitrary monadic operations, the essential idea of effect capabilities is to secure the operations of the layers in the monad stack using capabilities.

Concretely, this means that we define protected versions of monad transformers, and of the type classes associated to their effects, in which all the monadic operations are wrapped by the $CapT$ monad transformer. This way, while an external component can still access any layer of the monad stack using explicit lifting, it will not be able to perform operations on them unless it can present the required capability.

In particular, we define protected versions of the state and exception MTL transformers and their associated type classes. As a naming convention we append the P suffix to the name of the protected monad transformers and type classes. We now illustrate how to implement private and shared state (Section 4.1 and Section 4.2) and protected exceptions (Section 4.3).

```
class Monad m ⇒ MonadStateP c s m | m → s where
  getp :: (Capability c ⊃RW, p ⊃RW ReadPerm) ⇒ CapT (c p) m s
  putp :: (Capability c ⊃RW, p ⊃RW WritePerm) ⇒ s → CapT (c p) m ()
newtype StateTP c s m a = StateTP (StateT s m a) deriving ...
instance Monad m ⇒ MonadStateP c s (StateTP (c ()) s m) where
  getp  = lift ∘ StateTP $ get
  putp  = lift ∘ StateTP ∘ put
```

Fig. 2. Protected versions of the monad state type class and state monad transformer

4.1 Private Persistent State

Based on the state permission lattice (Section 3.2), we define the protected versions of the state monad transformer and corresponding type class (Figure 2). To use the *getp* function, one needs to have a capability c that implies the $ReadPerm$ read permission; and dually to use the *putp* function, one needs the capability that implies the $WritePerm$ write permission.

To illustrate, consider the following polymorphic $Queue$ using private state:

```
module Queue (enqueue, dequeue, QState ()) where
data QState p = QState p
instance Capability QState ⊃RW where
  attenuate (QState _) perm = QState perm
enqueue :: MonadStateP QState [s] m ⇒ s → m ()
enqueue s = do queue ← fromCapT (QState ReadPerm) getp
               fromCapT (QState WritePerm) $ putp (queue ++ [s])
dequeue :: MonadStateP QState [s] m ⇒ m s
dequeue   = do queue ← fromCapT (QState ReadPerm) getp
               fromCapT (QState WritePerm) $ putp (tail queue)
               return (head queue)
```

Thanks to the use of the $QState$ capability and the secure $MonadStateP$ class, the internal state of the queue is private to the $Queue$ module. Since the $QState$ data constructor is not exported, external access is prevented—even if explicit lifting can be used to access the respective instance of $MonadStateP$, it cannot be used to perform any monadic operation on it because the proper capability is required. We still require to export $QState$ as a type, in order to create a suitable monad stack, *e.g.* to instantiate an integer queue:

```
type M = StateTP (QState ()) [Int] Identity
```

To construct a monad stack we are only interested on the capability type, but not in any particular permission—however permissions will still be checked statically as required for each operation—hence we use () as the permission type in the definition of M.

4.2 Shared Persistent State

We now illustrate capability sharing with shared persistent state. We define a module $PriorityQueue$ that adds a notion of priority on top of $Queue$. In a priority queue one can access directly the most recent element having a high priority, using the $peekBy$ function. For efficiency, the $PriorityQueue$ module needs direct access to the internal state of the queue. As we do not want to do this by publicly exposing the capability $QState$, we send the capability on the channel $PQChan$ provided by $PriorityQueue$.

```
module Queue (enqueue, dequeue, QState ()) where
import PriorityQueue   -- to get PQChan channel
newtype QState p = QState p
instance Capability QState ⊃RW where
  attenuate (QState _) perm = QState perm
instance Send PQChan QState ReadPerm where
  receive perm = return $ QState perm
  -- enqueue and dequeue operations as before
```

The implementation of $PriorityQueue$ is as follows:

```
module PriorityQueue (PQChan (), peekBy) where
import Queue
data PQChan = PQChan
queueState :: QState ReadPerm
queueState = fromChannel PQChan $ receive ReadPerm
peekBy :: (Ord s, MonadStateP QState [s] m) ⇒ (s → s → Ordering) → m (Maybe s)
peekBy comp = do queue ← fromCapT queueState getp
                 if null queue then return Nothing
                               else return (Just $ maximumBy comp queue)
```

To use the internal state of the queue, the $PriorityQueue$ module imports $Queue$, defines and exports its channel $PQChan$, and retrieves the capability $QState$ with the read-only permission, as prescribed by $Queue$. $peekBy$ can access the internal state of the queue by using the $queueState$ capability and $fromCapT$.

4.3 Protected Exceptions

Exception handling may be seen as a communication between two modules, one that raises an exception, and one that handles it. For correctness or security reasons, we may wish to ensure that a raised exception can only be handled by specific modules. Protecting exception handling can also be achieved using exception capabilities. First, we define the private and public lattices, $\sqsupset_{Ex}$ and $\supset_{Ex}$, as permissions for throwing and catching exceptions, then in Figure 3 we define the protected versions of the standard $ErrorT$ monad transformer and $MonadError$ type class.

```
class (Monad m, Error e) ⇒ MonadErrorP c e m | c m → e where
  throwErrorp :: perm ⊃Ex ThrowPerm ⇒ e → CapT (c perm) m a
  catchErrorp :: perm ⊃Ex CatchPerm ⇒ m a → (e → m a) → CapT (c perm) m a
newtype ErrorTP c e m a = ErrorTP {runETP :: ErrorT e m a} deriving ...
instance (Monad m, Error e) ⇒ MonadErrorP c e (ErrorTP (c ()) e m) where
  throwErrorp       = lift ∘ ErrorTP ∘ throwError
  catchErrorp m h = lift ∘ ErrorTP $ catchError (runETP m) (runETP ∘ h)
```

Fig. 3. Protected versions of the monad error type class and error monad transformer

```
module ExLattice (⊃Ex, ThrowPerm, CatchPerm, TCPerm) where
data ThrowPerm = ThrowPerm
data CatchPerm = CatchPerm
data TCPerm    = TCPerm
  -- private lattice
class      a ⊐Ex b
instance a ⊐Ex a
  -- private instances
instance TCPerm ⊐Ex ThrowPerm
instance TCPerm ⊐Ex CatchPerm
  -- public lattice
class      a ⊐Ex b ⇒ a ⊃Ex b
instance a ⊐Ex b ⇒ a ⊃Ex b
```

Going back to our running example, we can make *dequeue* raise an exception when accessing an empty queue, in order to allow for recovery. Using a *QError* exception capability we can control which modules are allowed to define their own handlers:

```
module Queue (enqueue, dequeueEx, QState (), QError ()) where
  -- QState definition, type class instances and enqueue as before ...
data QError p = QError p
instance Capability QError ⊃Ex where
  attenuate (QError _) perm = QError perm
dequeueEx :: (MonadStateP QState [s] m, MonadErrorP QError String m) ⇒ m s
dequeueEx = do queue ← fromCapT (QState ReadPerm) getp
  if null queue then fromCapT (QError ThrowPerm) $ throwErrorp "Empty..."
                else do fromCapT (QState WritePerm) $ putp (tail queue)
                        return (head queue)
```

Recall from Section 2.4 the example of exception interference. Now the *consume* function can catch the exceptions it is interested in, while exceptions thrown by $dequeue_{Ex}$ will simply pass-through. Actually, it is not possible for *process* to catch those exceptions unless the *QError* capability is shared from the *Queue* module.

```
consume :: (MonadStateP QState [Int] m, MonadError String m,
            MonadErrorP QError String m) ⇒ m Int
consume = do x ← dequeue_Ex
             if (x < 0) then throwError "Process error"
                        else return x
process :: (MonadStateP QState [Int] m, MonadErrorP QError String m,
            MonadError String m) ⇒ Int → m Int
process val = consume `catchError` (λe → return val)
```

Consider a *debug* function in a module that has access to the $QError$ capability with permission implying $CatchPerm$. Then, *debug* can define a custom handler:

```
debug val = fromCapT (QError CatchPerm) $
              process val `catchErrorp` (λe → error "...")
```

Finally, for cases where the client is not trusted and cannot catch the exception, we can export a function $dequeue_{Err}$, that reraises the error using the $QError$ capability:

```
dequeue_Err :: (MonadStateP QState [s] m, MonadErrorP QError String m) ⇒ m s
dequeue_Err = fromCapT (QError CatchPerm) $ dequeue_Ex `catchErrorp` error
```

5 Related Work

Extensible effects (EE) [9] proposes an alternative representation of effects, in Haskell, that is not based on monads or monad transformers, and which can subsume the MTL library by providing a similar API. EE presents a client-server architecture where an effectful operation is requested by client code and is then performed by a corresponding *handler*. The internal implementation of EE uses a continuation monad, *Eff*, to implement coroutines, along with a novel mechanism for extensible union types. An effectful value has type *Eff* r where r is a type-level representation, based on the novel union types, of the effects currently available; thus defining a type-and-effect system for Haskell. EE does not describe any mechanism for restricting access to effects. Any effect available in the type-level tracking of effects is available to any component. To add two copies of the same effect while avoiding interference, the user is required to define a wrapper using a **newtype** declaration. This means that each effect can be uniquely identified by its type.

The *Effects* [1] library is an effect system implemented in the dependently-typed language Idris, based on algebraic effect handlers, also designed as an alternative to monads and monad transformers. Similar to EE, Effects keeps track of the available effects that can be used in an heterogeneous list. Performing an effectful operation requires a proof that the given effect is indeed available, but such proof is automatically generated if the effect is available. As with EE, Effects does not address the issue of controlling the access to effects. Any available effect can be used by any part of the system. References to copies of a same effect (*e.g.*two integer states) are disambiguated using labels in the effect-tracking list.

Effect capabilities are orthogonal to the mechanism used to implement effects. Recent mechanisms like EE and Effects focus on how to enable flexible composition of

effects—which is a well-known drawback of monad transformers—rather than on controlling access to them. We have shown how to apply effect capabilities to control access to effects in the context of monad transformers, mainly as a solution to known interference issues. We believe that the same approach should be applicable to other effect mechanisms, such as EE or Effects.

6 Future Work

We identify several venues for future work. A first one, regarding safety, arises from the fact that we have ignored a number of Haskell features that defeat the integrity of the type system. For instance, module boundaries can be violated using Template Haskell or the $GeneralizedNewtypeDeriving$ language extension, or generic programming. Recently, Safe Haskell [24] has been proposed as an extension to Haskell, implemented in GHC (as of version 7.2). Safe Haskell protects referential transparency and module boundaries by disabling the use of these unsafe features. Because the privacy of capabilities relies on effective module boundaries, we plan to integrate the effect capabilities library as an extension of Safe Haskell.

A second line of work aims to lower the amount of boilerplate code that is required, like the instances of $Capability$ and $Channel$ classes. This situation can be improved using *generic programming* (*e.g.*using the $GHC.Generics$ library), to provide default implementations for the $receive$ and $attenuate$ functions. This is already done in the downloadable implementation. A complementary approach is using TemplateHaskell [22], a template meta-programming facility for Haskell.

Another line of work concerns the integration of capabilities with tagged monads. In the model, each protected layer of the monad stack must be labeled with the capability namespace to which it is bound. This is similar to how a layer in a tagged monad setting must possess a tag in order to enable tag-directed type class resolution. The idea is to use the capability type constructor for both purposes at the same time; thus benefitting from the robustness with respect to the layout of the monad stack, provided by tagged monads, in addition to controlling access to each layer, using capabilities.

We also are interested in studying effect capabilities for other effects, like nondeterminism, concurrency, continuations, and particularly I/O. Currently, access to I/O operations through the IO monad is completely unrestricted. Using effect capabilities we plan to split access into several categories (*e.g.* file access, network access, etc.). Finally, it remains to be studied how effect capabilities can be provided in programming languages with imperative features but without explicit effects.

Acknowledgments. This work was supported by the Inria Associated Team REAL.

References

1. Brady, E.: Programming and reasoning with algebraic effects and dependent types. In: ICFP 2013 (2013)
2. Chang, B.-M., Jo, J.-W., Yi, K., Choe, K.-M.: Interprocedural exception analysis for Java. In: SAC 2001 (2001)

3. Coelho, R., Rashid, A., Garcia, A., Ferrari, F., Cacho, N., Kulesza, U., von Staa, A., Lucena, C.: Assessing the impact of aspects on exception flows: An exploratory study. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 207–234. Springer, Heidelberg (2008)
4. Fluet, M., Morrisett, G.: Monadic regions. Journal of Functional Programming (2006)
5. Gifford, D.K., Lucassen, J.M.: Integrating functional and imperative programming. In: Proceedings of the ACM Conference on LISP and Functional Programming (LFP 1986) (1986)
6. Harper, R.: Exceptions are shared secrets (December 2012), http://existentialtype.wordpress.com/
7. Harper, R.: Practical Foundations for Programming Languages. Cambridge U. Press (2012)
8. Hughes, J.: Global variables in Haskell. Journal of Functional Programming (2004)
9. Kiselyov, O., Sabry, A., Swords, C.: Extensible effects: An alternative to monad transformers. In: Haskell 2013 (2013)
10. Launchbury, J., Peyton Jones, S.: Lazy functional state threads. SIGPLAN Notices (1994)
11. Levy, H.M.: Capability-based computer systems, vol. 12. Digital Press Bedford, Massachusetts (1984)
12. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: POPL 1995 (1995)
13. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis (2006)
14. Milner, R., Harper, R., MacQueen, D., Tofte, M.: The Definition of Standard ML. MIT Press (1997) (revised)
15. Moggi, E.: Notions of computation and monads. Information and Computation (1991)
16. Piponi, D.: Tagging monad transformer layers (2010), http://blog.sigfpe.com/2010/02/tagging-monad-transformer-layers.html
17. Robillard, M., Murphy, G.: Static analysis to support the evolution of exception structure in object-oriented systems. In: TOSEM 2003 (2003)
18. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems (1975)
19. Sangiorgi, D., Walker, D.: The Pi-Calculus: A Theory of Mobile Processes. Cambridge U. Press (2003)
20. Schaefer, C.F., Bundy, G.N.: Static analysis of exception handling in Ada. Software—Practice and Experience 23(10), 1157–1174 (1993)
21. Schrijvers, T., Oliveira, B.C.: Monads, zippers and views: virtualizing the monad stack. In: ICFP 2011 (2011)
22. Sheard, T., Jones, S.P.: Template meta-programming for haskell. SIGPLAN Not. 37(12), 60–75 (2002)
23. Snyder, M., Alexander, P.: Monad factory: Type-indexed monads. In: Page, R., Horváth, Z., Zsók, V. (eds.) TFP 2010. LNCS, vol. 6546, pp. 198–213. Springer, Heidelberg (2011)
24. Terei, D., Marlow, S., Peyton Jones, S., Mazières, D.: Safe Haskell. In: Haskell 2012 (2012)
25. Wadler, P.: The essence of functional programming. In: POPL 1992 (1992)

Fusion: Abstractions for Multicore/Manycore Heterogenous Parallel Programming Using GPUs

Anderson Boettge Pinheiro, Francisco Heron de Carvalho Junior, Neemias Gabriel Pena Batista Arruda, and Tiago Carneiro

Mestrado e Doutorado em Ciência da Computação,
Universidade Federal do Ceará, Brazil
{ab.pinheiro,heron,gabrielpb,carneiro}@lia.ufc.br

Abstract. Graphic processing units (GPU) have been consolidated as general-purpose computational devices that combine a challenging programming model with an impressive acceleration of HPC programs whose total execution time are dominated by performance-critical small sections of code. However, there is still a lack of high-level programming language abstractions for better specifying heterogenous parallel computations using these devices. This paper proposes Fusion, an extension of Java that introduces new abstractions for heterogenous multicore-GPU programming, taking advantage of new features introduced by the NVIDIA's Kepler architecture, such as Hyper-Q and Dynamic Parallelism.

1 Introduction

In the recent years, the use of Graphics Processing Units (GPUs) [16] have been consolidated for general purpose computing, for accelerating sections of code that exhibit high computational costs in programs with high performance requirements. GPUs now represent a class of computational accelerators that have been incorporated in a number of high performance computing platforms. Other important classes are FPGAs (Field-Programmable Gate Arrays) [11] and MICs (Many Integrated Cores) [7]. The success of GPGPU[1] computing may be measured by the large number of papers in the literature about the design of parallel algorithms that may use them efficiently, produced mostly by researchers from computational sciences, engineering and computing science.

However, the initiatives on new programming abstractions that aims at simplifying the description of these algorithms on GPUs, without introducing significant performance overheads, are still incipient. The programmer still needs specific knowledge about the peculiarities of the target GPU architecture, as well as programming techniques that are complex even for parallel programmers with large experience. The existing programming interfaces with full expressiveness for taking advantage of GPU architectures efficiently, such as CUDA and

[1] General-Purpose Graphic Processing Units.

F.M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 109–123, 2014.

OpenCL, still works at a lower level of abstraction, requiring that programmers have knowledge about a number of technical details about the GPU architectures, as well as some undocumented programming tricks. Attempts of including support for GPGPU programming in high-level object-oriented languages, such as Java, still relies on providing direct connections to the CUDA architecture.

We are particularly interested in proposing and evaluating new programming abstractions for heterogenous multicore/manycore parallel computing, where a set of parallel threads running in a multicore processor wants to accelerate a parallel computation by using a GPU device, shared among them. This is an interesting problem not only for high-end HPC platforms, since almost all processors are multicore, equipping a wide range of machines, from low-cost smartphones to high-end server workstations. In the recent years, NVIDIA, one of the main GPU providers, launched the Kepler architecture, including the support for the Hyper-Q and Dynamic Parallelism (DP) extensions, which offer new opportunities for increasing the expressiveness of parallel programming interfaces on describing other known patterns of parallel computation using these devices. Also, they attempt to remove certain bottlenecks that makes inefficient the connection between multi-thread parallel programs in the host processor and the GPU device, aiming at attending heterogenous parallel computing requirements.

This work aims at proposing new parallel programming abstractions in a new object-oriented programming language based on Java, so-called Fusion[18], for expressing heterogenous multicore/manycore parallel computations, offering a higher level of abstraction compared to the existing alternatives, but still offering to the programmer total control over the usage of the device resources. The implementation of the abstractions of the proposed language is possible due to the expressiveness offered by the Kepler architecture, with Hyper-Q and DP.

This paper is structured in five more sections. Section 2 presents an overview about the GPU architecture and related works on programming languages for GPGPU programming. Section 3 presents the Fusion programming language, with focus on the abstractions it introduces for heterogenous multicore/manycore parallel programming. Section 4 presents a case study of program developed with Fusion, for evaluating it in practice. Finally, Section 5 presents the conclusions of this work, also presenting lines for further works.

2 Context

Graphics processor units (GPU) have emerged in the late 1970s as coprocessors providing fixed graphics functions organized in a pipeline. In the late 1990s, GPUs with programmable vertex and pixel shading pipeline steps, through assembly and shader languages, such as OpenGL's GLSL, NVidia's CG and Microsoft's HLSL [2] have motivated their use for general-purpose computations, consolidated in the 2000's, starting the interest in designing General-Purpose Graphics Processing Units (GPGPUs) and GPU Computing [15]. In 2006, NVIDIA released the G80 architecture [12], introducing GPUs with general-purpose processors so-called *stream processors* (SP), as well as the CUDA general-purpose parallel programming platform, for exploiting their performance.

CUDA frees programmers from many complex tasks, such as scheduling threads, hiding the GPU architecture behind an API. Together with G80's high performance and availability, CUDA rapidly popularized GPGPU. GPU computing is now an attractive research field, exploiting algorithm development for effectively accelerating critical portions of the code called kernels [16,8].

In 2009, NVIDIA released the *Fermi* architecture [10], introducing full C++ support, double-precision float-point operations, ECC support, cache hierarchy and parallel kernel execution from multiple streams. *Kepler* was released in 2012 [5], introducing two new features of our interest in this paper: *Dynamic parallelism* (DP) and *Hyper-Q*.

2.1 An Overview of Kepler GPU Architecture

In CUDA, the code that runs in the CPU, so-called *host*, can launch code that will execute in the GPU, so-called *device*. The host and the device are connected through a PCI Express bus. The code processed by the device is called *kernel*.

When a kernel is launched, threads are grouped in *blocks*, organized in a *grid*. Blocks are divided into groups of 32 threads, so-called *warps*. Each warp is scheduled to an SIMD processor containing 192 single-precision streaming processors (SP) and 64 double-precision unities, so-called SMX (Next-generation Streamming Multiprocessor). The full Kepler GK110 GPU has 15 SMX.

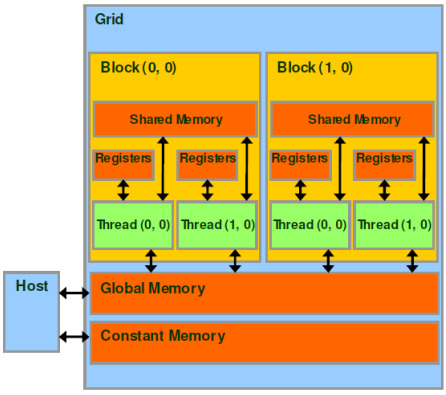


Fig. 1. High level architecture of an NVIDIA GPU [12]

The memory hierarchy in a Kepler GPU has several levels. The most important are *global*, *shared* and *local* memories. The communication between host and device occurs through the global memory. The global memory is the biggest and slowest memory. Each thread has its local memory and threads within a block communicate through the shared memory. Local and shared memory are the fastest memory, but they are small. In Kepler, as in Fermi, each SMX has 64 KB of onchip memory, split into shared memory and L1 cache [3]. A high level description of this organization can be seen in Figure 1.

Kepler introduced Dynamic Parallelism (DP) for launching new kernels from kernels in execution, making it possible the GPU code generate more work to the GPU, for increasing the granularity in some critical regions of the code. In turn, Hyper-Q enables the scenario where multiple CPU threads launch cooperative kernels simultaneously, through independent *streams*. Kepler GPUs supports 32 streams [3]. This feature is useful in heterogenous parallel computing, where a multithreading parallel program in the CPU want to use GPU cooperatively, without synchronizing for launching a single kernel, as in Fermi.

DP and Hyper-Q provide more expressiveness to represent patterns of parallel computations in GPUs. The aim of this work is to exploit them through new abstractions in a proposed object-oriented programming called Fusion.

2.2 Related Work

There is a number of proposals of GPU programming interfaces that we consider relevant in the context of our work with Fusion. They are implemented on top of existing languages, mainly C, C++, Fortran, Python and Java.

Other approaches rely on the direct incorporation of CUDA abstractions to the language, such as CUDA Fortran [20] and JCuda [21], without proposing new high-level abstractions for GPU programming. In the case of JCuda, the abstractions of object-oriented programming, inherited from Java, may help to encapsulate the complexity of GPU programming. JaBEE [22] and Rootbeer [19] are object-oriented interfaces for GPU Programming, based on Java. However, they do not exploit the new possibilities due to Hyper-Q and DP neither introduce specific programming abstractions. However, Rootbeer shows that it is possible making a transparent translation from high-level abstractions to CUDA with acceptable performance overheads. Finally, the Lime Language, a Java based high level language for targeting heterogenous systems, that allows an optimizing compiler to generate GPU code [1,6].

3 The Fusion Language

We propose the Fusion programming language to face the challenging requirements of heterogenous multicore/manycore parallel programming[18]. The premises that guided the design of Fusion are discussed in the next paragraphs.

Separation of Concerns from the User Perspective. We are interested in separating the burden of programming concerns related to GPU programming from the high-level application concerns. For that, we assume the existence of two kinds of programmers, viewed from the application perspective: *developers* and *specialists*. Developers have knowledge about GPU architectures and the best parallel programming techniques (and tricks) for them, being able to produce highly tuned code for a given computation that must be accelerated using a GPU. In turn, specialists have knowledge about the application, being able to decide which computations are critical for the application performance and must

be deployed in the GPU for execution. We are looking for the successful level of programming abstraction of domain-specific scientific computing libraries, where the developers must know how to implement the best algorithms and parallelism strategies, removing this burden from the specialists, which are now only concentrated on identifying where and when the library functionality must be used.

Object-Orientation. Object-oriented languages have been consolidated as the most influential paradigm in most of the application domains since the 1990s, in such a way that modern software engineering techniques have been mostly based on this paradigm. Thus, object-orientation makes it possible to reach both a large spectrum of users and modern software engineering practices. In the last decade, C++, Java and C# have been applied even in the implementation of scientific computing programs, despite the performance overheads due to their abstractions. We adopt Java as a basis for Fusion, due to its dissemination, portability and recent gains in performance due to JIT (*Just-In-Time*) compilation. GPU programming code may be encapsulated into objects, for separating the concerns of developers and specialists and allowing for JIT compilation of GPU objects directly to the GPU code, bypassing virtual execution.

Kepler Architecture. The interest in developing Fusion has been highly motivated by the inception of the Kepler architecture, supporting Hyper-Q and Dynamic Parallelism (DP) technologies. They have removed some severe restrictions of GPU programming, making it possible to express more patterns of parallel computation and better integration with multi-core parallelism in the host processor, an important tendency in heterogenous paralell programming. Most of the linguistic abstractions proposed by Fusion on top of Java takes into consideration the expressive power due to the Kepler architecture.

The abstractions of the Fusion programming model are presented in Section 3.1, together with the description of its syntax by using the illustrative example presented in Section 3.2.

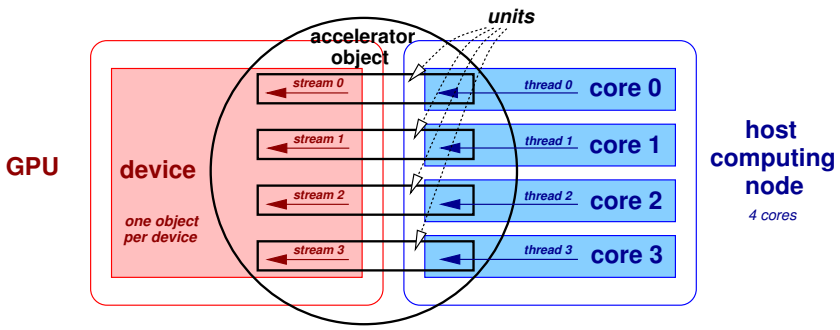


Fig. 2. Linking a GPU and a Computing Node Through an ***Accelerator Object***

3.1 The Abstraction of Accelerator Objects - Concepts

Fusion extends Java with the abstraction of *accelerator objects*, a special kind of object whose methods may be kernels that execute in a GPU. So, the Fusion JIT compiler will have the ability to generate native GPU code for these methods. In what follows, we scrutinize the structure of accelerator objects and how they are specified by means of *accelerator classes*.

An accelerator object is a collection of *units*, each one linking a thread of the host object to a stream of an accelerator device of the host node (Figure 2). Each unit carries a subset of the state (attributes) and methods of the accelerator object. The abstraction of units is a way of dealing with heterogenous parallel computing, involving multicore processors and GPUs connected to them. This is the main motivation for the Hyper-Q technology in the Kepler architecture.

An attribute of an accelerator object may be placed in either the host memory or the global memory of the device. In turn, a method may be either a *host method* or a *kernel method*. Host methods will execute in the host, whereas kernel methods will execute in the device, over the stream associated to the unit. A host method may access only host attributes, declared outside any unit scope. In turn, a kernel method may access only the attributes of its unit.

A *parallel method* is a method that belongs to a team of units. They must be invoked by all the host threads that are associated these units for completing the invocation. A method that is not parallel, i.e. that belongs to a single unit, is called a *singleton method*.

3.2 The Abstraction of Accelerator Classes - Syntax

The main extension of the Java syntax consists of a set of reserved words used in the definitions of accelerator classes, i.e methods, attributes and operations in the memory structure (access, allocations and types) on the GPU device. In Table 1 there is a description of keywords added to Java syntax.

Table 1. Description of keywords added in Java syntax

Keyword	Description
`accelerator`	Qualifier for accelerator classes.
`kernel, device and host`	Access modifier for methods.
`unit`	Quantifier for units on accelarator classes.
`parallel`	Access modifier for methods and units.
`async`	Access indentifier for memory operations
`shared, global, constant, texture`	Access modifiers for memory access

We have another set of constructors associated to kernel method that indicates the map of threads to GPU: `threads` $\lll t_x, t_y, t_z \ggg$; `blocks` $\lll b_x, b_y, b_z \ggg$, where each axis is represented by a tuple index, and another that indicates the device number where the accelerator object will be executed, `idAcceleratorObject` $\ll device \gg$.

Moreover, the mapping of threads in kernel method is analog to CUDA language, we have an set of built-in variables that reveal information on grid mapped in the device for a specific axis coordinate (*axis*): `blockSize(axis)` indicate the number of threads in block; `gridSize(axis)` indicate the number of blocks; `threadIdx(axis)`, indicate the index of thread in block; `blockIdx(axis)`, indicate the index of block in grid. Accelerator classes specifies the state and interface of accelerator objects. We will use the pseudocode in the Listing 1.1 for illustrating their syntax.

Listing 1.1. Implementation of the Parallel Unit `enumerate` and the kernel method `dfs`

```
1  accelerator class EnumAccel{
2    global int path_d;
3    int path_h;
4    ...
5    EnumAccel(<args>){ ... }
6    synchronized public int getUpper_Bound(){ ... }
7    public kernel dfs (int upper_bound) threads<192>
8      blocks<16>{ ... }
9    parallel unit Enumerate{
10     setData(int path_h){
11       async path_d = path_h;
12     }
13   public kernel dfs (int upper_bound) threads<192>
14     blocks<nPreFixos / numThreads(x)
15       + (nPreFixos % numThreads(x) == 0 ? 0 : 1)>{
16      int tx = threadIdx.x; ty = threadIdx.y;
17      int bx = blockIdx.x; by = blockIdx.y;
18      shared float <id_variable>;
19      ...
20     }
```

The `EnumAccel` class specify one unit declaration, called `Enumerate`. The former has host methods `setData`, for receiving the operands and correctly placing them in the global memory of the device, and `getResult`, for fetching the result from the device and returning it to the caller host object.

The `parallel` keyword in the `Enumerate` unit declares it as a *parallel unit*. A parallel unit represents an homogeneous set of units with the same attributes and methods. The host object may instantiate as many units of a parallel unit it desires, for distinct host threads. The `setData` method will copy partitions of the input matrices to the unit attributes of each `Enumerate` unit.

A method whose signature is declared in the scope of an accelerator class, so-called *class method*, must have an implementation in the scope of each unit. In turn, methods declared in units are so-called *unit methods*. A class method may declare a body, representing a default implementation for units that do not provide a specific implementation, which can access only host attributes.

The `parallel` keywork may also declare a *parallel method*. Clearly, parallel methods must not be declared in the scope of singleton units. Kernel methods may invoke device methods, declared using the `device` modifier, and other kernel methods, which is now possible in Kepler architectures by using dynamic parallelism (DP) technology. Kernel methods are directly compiled to CUDA intermediate code.

Attributes declared in the scope of the class are host attributes, whereas attributes declared in the scope of units are unit attributes. Local variables declared inside kernel methods may be placed in any place in the memory hierarchy of the GPU. By default, they are placed in the local memory. The programmer may force a variable to be placed in other place by using the modifiers `global`, `shared` and `constant` and `texture`, such as in CUDA.

3.3 Instantiation of Accelerator Objects

The instantiation of an accelerator object is a collective operation involving all the host threads. Firstly, it is instantiated using the `new` operator, outside their scope, like in the following example:

`EnumAccel` *objAccel* `= new EnumAccel<`*device*`>(···)`

instantiates an accelerator object *objAccel* from the accelerator class `EnumAccel`, where *device* is an optional integer value specifying the device where the accelerabor object will be placed. If *device* not informed, it will be placed in a free device, if one exists.

After instantiation, the methods of the units of *objAccel* cannot still be accessed. They must be initialized, using the `unit` declaration, in each host thread. For instance, the initialization of a parallel unit of *objAccel* identified by `Operation` in each host thread that will hold a unit of *objAccel* has the following syntax:

`EnumAccel:Enumerate` *objAccel_unit* `= unit Enumerate<`*size*`> of` *objAccel*`;`

, where *size* is the number of `Operation` units. Notice that this code must be executed by each host thread. After that, the unit operations may be accessed through the unit variable *objAccel_unit*. An access to a method of a unit of a parallel unit before all units have been initialized will block until the initialization is complete. For singleton units, the *size* parameter is not necessary.

3.4 Discussion

Accelerator objects make it possible to hide most of the complexity of heterogenous multicore/GPGPU programming through encapsulation. Indeed, the abstraction of units make transparent the configuration of parallel streams in GPU devices that follows the Kepler architecture. In the example, each unit of the `Enumerate` parallel unit, connected to a host thread, is associated to a different stream, in such a way that kernels invoked through each unit run in parallel. A programmer that wants to take advantage of the processing power of a GPU to perform a computation must only concern with instantiating accelerator objects from the accelerator classes that implements the desired functionalities and invoking their parallel methods to perform calculations. In turn, the programmer of accelerator classes will have all the expressive power of CUDA to communicate with the GPU as efficient as possible, implementing highly tuned

algorithms for taking advantage of GPU computation power. We argue that Fusion provides important gains in abstraction without loss in efficiency, since its abstractions have simple mapping to CUDA constructors, promoting better programmer productivity for developing code for GPUs.

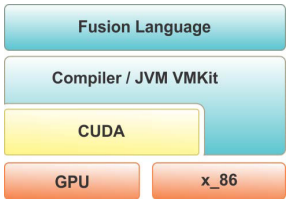


Fig. 3. The Fusion Prototype Architecture

3.5 Prototypic Implementation

Since Fusion is an extension to Java, we have looked for an existing Java virtual machine (JVM) with support for connection with CUDA for building its first prototypic implementation. We found J3 [9], a JVM implementation built on a substrate of the LLVM (Low Level Virtual Machine) project [14][13], so-called VMKit. The JIT compiler of J3 is the LLVM compiler, which makes it possible the addition of new features through intrinsic functions, ensuring that the functionalities needed for supporting the GPU architecture can be inserted into the JVM without the need for modifying the J3 compiler.

PTX (Parallel Thread Execution) is a low-level intermediate code generated by CUDA compilers for the GPU architectures. We chose the LLVM compiler due to its ability to generate intermediate code that is similar to PTX code. The code generated by the LLVM compiler is called LLVM IR, which is later translated into PTX. With the PTX code, we can call the execution compiler of CUDA architecture for generating the machine code for the target GPU architecture.

Figure 3 outlines the proposed architecture. The Fusion abstractions are at the top level, on top of the modified VMKit, the virtual machine of J3. The CUDA layer is responsible for the interface between the Fusion-J3 compiler and the device. It is necessary because the Fusion-J3 compiler, which has the responsibility in generating the intermediate code, do not generate the specific machine code of the device. This is the role of the CUDA compiler.

The use of a CUDA compiler is the most important restriction of the first prototypic implementation of Fusion, since it restricts its use to NVIDIA devices. Further works will remove this restriction by allowing a compiler to translate into the machine code of AMD and ATI devices, possibly using OpenCL.

4 Case Study: Complete Enumeration in ATSP

The Travelling Salesman Problem is stated as follows: find the shortest route for a travelling salesman who wants to visit a list of cities, each city exactly

once, and return to the initial city. It is one of the most important combinatorial optimization problems. If, for all a a and b, the distance from a to b is equal to the one from b to a, then the TSP is symmetric. Otherwise, it is asymmetric.

The application chosen for this case study is a complete enumeration of Asymmetric Traveling Salesman Problem (ATSP) instances, based on a GPU-based schema for Depth-First Branch-and-Bound (DFS-B&B) algorithms proposed by Carneiro et al. [4].

The B&B method is an important class of algorithm in combinatorial optimization, since it is the main method for solving hard combinatorial optimization problems in optimality. All B&B algorithms consist of three phases [23]: *Branching*, the partition of a problem into smaller problems; *Bounding*, evaluation of partial and final solutions, by using upper and lower bounds; and *Pruning*, the act of eliminating large portions of the search space. The complete enumeration herein presented is a DFS-B&B with no prune rules, i.e., a complete enumeration of all solutions of an ATSP instance. The complete enumeration of an ATSP instance is a computationally intensive algorithm, since an ATSP instance with N cities has $(N-1)!$ solutions.

The parallelization strategy adopted by Carneiro et al. (2011) is called Collegial B&B. In this strategy, several independent search procedures are performed in parallel. The algorithm is initialized by a modified serial DFS-B&B procedure that performs B&B serially until a specific depth in the solutions space is reached. The current path in B&B tree is saved as a node into the *Active Set*, a set that contains nodes that have been evaluated but not yet branched. Each node constitutes a root to the DFS-B&B kernel, which is responsible for finding the best local solution in a ramification. This process generates a significant amount of threads, aiming at achieving high GPU occupancy.

As shown in Listing 1.2 (lines 1 to 4), the use of streams enable parallelism, ensuring that the roots will be calculated in parallel by the kernels DFS-B&B, each one executing over an independent stream. Before the launching of each DFS-BB kernel, between the lines 6 and 23, the data is transferred to the device, by indicating the appropriate stream, where the kernel will be launched, in the transference of the corresponding portions of each kernel. The number of threads is calculated before the invocation of the DFS kernels, being dependent on the number of created roots.

Listing 1.2. Stream Creation and Launching of the DFS-BB Kernels [17]

```
1  cudaStream_t vectorOfStreams[numStreams];
2
3  for(int stream_id=0; stream_id<numStreams; stream_id++)
4    cudaStreamCreate(&vectorOfStreams[stream_id]);
5
6  for(int stream_id=0; stream_id<numStreams; stream_id++)
7        cudaMemcpyAsync(&path_d[stream_id*chunk*nivelPreFixos],
8                        &path_h[stream_id*chunk*nivelPreFixos],
9                        qtd_threads_streams[stream_id]*sizeof(short)*nivelPreFixos,
10                       cudaMemcpyHostToDevice,vectorOfStreams[stream_id]);
11
12 for(int stream_id=0; stream_id<numStreams; stream_id++)
13       cudaMemcpyAsync(&melhorSol_d[stream_id*chunk],
14                       &melhorSol_h[stream_id*chunk],
15                       qtd_threads_streams[stream_id]*sizeof(int),
16                       cudaMemcpyHostToDevice,vectorOfStreams[stream_id]);
17
18 for(int stream_id=0; stream_id<numStreams; stream_id++)
19       cudaMemcpyAsync(&sols_d[stream_id*chunk],
```

```
20                          &sols_h[stream_id*chunk],
21                          qtd_threads_streams[stream_id]*sizeof(int),
22                          cudaMemcpyHostToDevice, vectorOfStreams[stream_id]);
23
24  for(int stream_id=0; stream_id<numStreams; stream_id++)
25    dfs_cuda_UB_stream<<<num_blocks, block_size,0, vectorOfStreams[stream_id]>>>
26                    (N, qtd_threads_streams[stream_id], mat_d,
27                     &path_d[stream_id*chunk*nivelPreFixos], nivelPreFixos,
28                     999999,&sols_d[stream_id*chunk], &melhorSol_d[stream_id*chunk]);
```

In the Fusion code, an accelerator class so-called `EnumerationAccel` is specified, with a single parallel unit so-called `Enumerate`, which implements the kernel method `dfs`, representing the DFS-B&B kernel. The code of the unit `Enumerate` is presented in the Listing 1.3. The kernel method `dfs` is similar to the CUDA kernel of the related work. For this reason, this code is omitted from Listing 1.3 (ellipsis in line 14).

Listing 1.3. Implementation of the Parallel Unit `enumerate` and the kernel method `dfs`

```
1  parallel unit Enumerate
2  {
3    public void setData(int[] path_h)
4    {
5      async path_d = path_h;
6      async melhorSol_d = melhorSol_h;
7      async sols_d = sold_h;
8    }
9    public kernel dfs (int upper_bound) threads<192>
10     blocks<nPreFixos / numThreads(x) + (nPreFixos % numThreads(x) ==
11     0 ? 0 : 1)>{...}
12   }
13 }
```

In a Fusion multi-thread computation, the parallel units of the accelerator object are instantiated by the host threads. In the Listing 1.3, it is possible to observe that each unit perform its own asynchronous data copies to the device (lines 5, 6 and 7), using the usual assignment operation with the `async` modifier. The units are instantiated through the set of host threads created and instantiated by the application. The creation of a set of threads for a specific object may be seen in the Listing 1.4, where `size` indicates the number of *threads*. Notice that the accelerator object is mapped to the device 0.

Listing 1.4. Instantiation of an accelerator object from `EnumerationAccel`

```
1  enumeration = new EnumerationAccel<0> (chunk, nPreFixos, N, nivelPreFixos,
2    mat_h, path_h, melhorSol_h, sols_h, size, qtd_threads_streams);
3  ForkJoinPool forkJoinPool = new ForkJoinPool(size);
4  forkJoinPool.invoke(new TaskEnumeration(this, Enumeration) );
5  otimo_global = enumeration.getMelhorSol();
```

The instantiation of units occurs inside each thread of the created set. In the Listing 1.5, we can see the instantiation in line 9, as well as the call to the parallel kernel method in line 16. Remember that the method will be executed when all the `size` units will be instantiated.

Listing 1.5. Initialization of the Unit `Enumerate`

```
1  public class TaskEnumeration extends RecursiveTask<Double>
2  {
3    private EnumerationAccel accel_obj;
```

```
4     private int rank;
5
6     TaskEnumeration(EnumerationAccel accel_obj){
7       super();
8       this.accel_obj = accel_obj;
9     }
10    protected Object compute(){
11      private EnumerationAccel:Enumerate accel_obj_unit;
12      accel_obj_unit = unit Enumerate<size> of accel_obj;
13      accel_obj_unit.dfs(999999);
14      return null;
15    }
16  }
```

4.1 Performance Evaluation

This performance evaluation was designed with two questions in mind:

1. Can a Java developer obtain better performance by using Fusion for accelerating critical code fragments ?
2. Can a CUDA developer adopt a host language with higher level of abstraction without significant performance penalties ?

For that purpose, four program variants were created from Carneiro et al. (2011): `CUDA (A1)`, adapted for the use of streams; `CUDA_OMP (A2)`, adapted for the use of OpenMP threads, where each thread creates a stream and launches a kernel; `Java Sequential (A3)`, where all operations are performed sequentially; `Java Parallel (A4)`, based on `A2`, with the solution space partitioned across the host threads, by creating a Java thread for each workload; and `Fusion model(A5)`, based in `A1` and `A4`, that presents the execution model of Fusion.

In the `Java Parallel` version, the partitioning of the workload is given by the number of processing cores, in order to achieve a maximum occupancy. This partitioning strategy is also adopted in algorithms `CUDA`, `CUDA_OMP` and `Fusion model`. In `Fusion model`, each host thread creates a parallel unit that will launch a kernel implemented in CUDA. The interface between Java and CUDA is performed via JNI (Java Native Interface). Every parallel unit is implemented as an inner class in the class that represent the accelerator object.

In the experiments, each program repeated the complete enumeration by 32 times. The higher and the lower execution times have been discarded. The execution time has been measured by using the Linux function `time`. ATSP instances were constructed by using glibc *rand* function. These instances represent complete graphs, where each element r_{ij} belonging to the R_{NxN} cost matrix can vary from 0 to 1000. In Table 2, we can see the execution times for each implementation of ATSP instances. In Table 2, we present the time required by each implementation to perform the complete enumeration. The experimental environment has the following characteristics:

1. Operating system: Linux Ubuntu 13.04 32 bits;
2. Processor: Intel core i5-3550 3.30GHz with four cores and 8 GB Ram;
3. Accelerator: GeForce Nvidia GTX690, 2 GB, 1536 cuda cores.

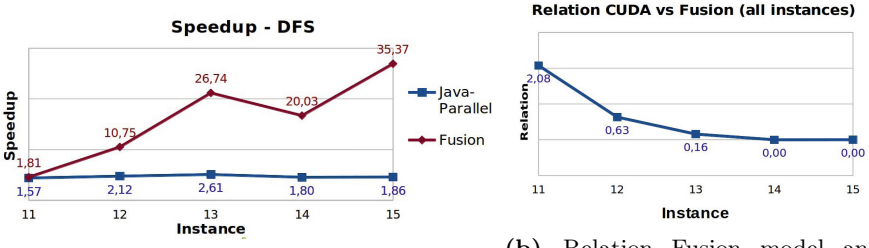
For all instances, the execution times of Fusion are better than the execution times of the Java multithreaded version, as expected. This is more evident for higher levels. In relation to the CUDA-C versions, Fusion is equivalent with the `CUDA_OMP` version for instance 12. In turn, it is equivalent to the `CUDA` version in higher instances, with significant results for instances 14 and 15.

Table 2. Execution time for each implementation of complete enumeration

Time Milliseconds					
Instance	A1	A2	A3	A4	A5
11	51.75	58.95	292.80	186.30	161.55
12	130.47	229.42	2290.43	1078.50	212.97
13	873.40	1617.97	27032.30	10351.10	1010.83
14	17877.60	20345.50	357397.75	190260.75	17842.60
15	168821.03	220479.32	5974113.6	3219878.85	168926.83

In Figure 4a, it is possible to observe the speedups for `Java Parallel` and `Fusion` versions, compared with the serial version of the same algorithm. The results demonstrate that Java programmers can benefit from the abstractions proposed by Fusion. We can observe that the use of accelerators objects, in this example, provided a significant performance gain especially for larger instances reaching the order of 35 times faster than serial version.

It is observed a drop in speedup for the `Fusion model` in the transition between instances 13 and 14. This pattern was also observed in the CUDA-C versions. In a more specific analysis, we identified that parallel execution of all streams on the GPU device occurs only until the instance 13. For larger instances, this is not possible. Serialization of streams occurs on the device due to the large amount of data required for each workspace solutions.



(a) Speedup Java Parallel and Fusion model **(b)** Relation Fusion model and CUDA

Fig. 4. Speedup and relation between programs

The `CUDA` speedup is not presented since the serial version is written in Java, for an evaluation of the weight of Java abstractions. However, the analysing of the Figure 4b also shows that the CUDA programmers can use Fusion abstractions without loss of performance. Note that the instance 11, `CUDA` is 2x faster than `Fusion`, but this difference is reduced in other instances. In instances 14 and 15 the relation is zero demonstrated that the execution time Fusion is equivalent

to CUDA. The results show that Fusion model accomplishes its initial purpose: combine high-level abstractions with high performance.

5 Conclusions and Lines for Further Works

Fusion is an extension of Java with abstractions for heterogenous multicore/-manycore parallel computing. It introduces a special kind of object, so-called accelerator object, for taking advantage of object-orientation for hiding the complexity of GPGPU architecture from the programmers. It also takes advantage of recent advances in the GPU architecture for implementing new abstractions, due to the introduction of Hyper-Q and Dynamic Parallelism technologies with Kepler architecture, extending the programming model of GPUs.

Other Java based approaches for GPGPU programming are mostly concerned with providing a direct connection to the CUDA architecture, only interested in bringing object-orientation to GPGPU programming in its usual form. This is the main difference with respect to Fusion, which also introduces new linguistic abstractions to Java, making it possible the evaluation of this alternative.

The case study presented in Section 4 demonstrates that Fusion may reduce the complexity of the GPGPU code in relation to the same code written in CUDA-C, hiding architecture specific details under its abstractions. For instance, the management of independent streams, a key concept introduced by the Kepler architecture for heterogenous parallel computing, is completely hidden behind the abstraction of units. In fact, all the complexity of the GPU architecture is confined in the accelerator object EnumerationAccel. Only the programmer of the accelerator object is responsible to tune the performance of the computation with the characteristics of the target GPU.

Future works will complete the implementation of the Fusion compiler. It is still necessary to study how to reconcile certain key features of Java, derived from its virtual execution nature, with GPU programming. The most challenging is dynamic memory management. In fact, the Java garbage collector is not able to reach data placed in the memory hierarchies of the GPU, requiring explicit deallocation routines like in CUDA-C.

References

1. Auerbach, J., Bacon, D.F., Cheng, P., Rabbah, R.: Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In: ACM Sigplan Notices, vol. 45, pp. 89–108. ACM (2010)
2. Blythe, D.: The Direct3D 10 System. ACM Transactions on Graphics (TOG) 25(3), 724–734 (2006)
3. Bradley, T.: Hyper-Q Example. NVidia Corporation. Whitepaper v1.0 (2012)
4. Carneiro, T., Muritiba, A.E., Negreiros, M., de Campos, G.A.L.: A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU. In: 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 41–47 (2011)
5. Corporation, N.: Nvidia's Next Generation CUDA Compute Architecture: Kepler GK110. Whitepaper v1.0 (2012)

6. Dubach, C., Cheng, P., Rabbah, R., Bacon, D.F., Fink, S.J.: Compiling a High-Level Language for GPUs (Via Language Support for Architectures and Compilers). In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, pp. 1–12. ACM, New York (2012)
7. Duran, A., Klemm, M.: The Intel® Many Integrated Core Architecture. In: 2012 International Conference on High Performance Computing and Simulation (HPCS), pp. 365–366. IEEE (2012)
8. Fatahalian, K., Houston, M.: A Closer Look at GPUs. Commun. ACM 51, 50–57 (2008)
9. Geoffray, N., Thomas, G., Lawall, J., Muller, G., Folliot, B.: VMKit: A Substrate for Managed Runtime Environments. In: Virtual Execution Environment Conference (VEE 2010), Pittsburgh, USA. ACM Press (March 2010)
10. Glaskowsky, P.N.: Nvidias Fermi: The First Complete GPU Computing Architecture. NVIDIA Corp. (2009)
11. Herbordt, M.C., VanCourt, T., Gu, Y., Sukhwani, B., Conti, A., Model, J., DiSabello, D.: Achieving High Performance with FPGA-Based Computing. Computer 40(3), 50 (2007)
12. Kirk, D., et al.: Nvidia CUDA Software and GPU Parallel Computing Architecture. In: ISMM, vol. 7, pp. 103–104 (2007)
13. Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL (December 2002)
14. Lattner, C.: The LLVM Compiler Infraestructure (September 2012), `http://www.llvm.org`
15. Nickolls, J., Dally, W.J.: The GPU Computing Era. IEEE Micro 30(2), 56–69 (2010)
16. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU Computing. Proceedings of the IEEE 96(5), 879–899 (2008)
17. Pessoa, T.C.: Estratégias Paralelas Inteligentes para o Método Branch-and-Bound Aplicadas ao Problema do Caixeiro Viajante Assimétrico. Master's thesis, Universidade Estadual do Ceará (2012)
18. Pinheiro, A.B.: Abstrações Linguísticas para Programação de Propósito Geral sobre Aceleradores Computacionais Baseados em GPU. Master's thesis, Universidade Federal do Ceará (2013)
19. Pratt Szeliga, P.C., Fawcett, J.W., Welch, R.D.: Rootbeer: Seamlessly Using GPUs from Java. In: 14th IEEE International Conference on High Performance Computing and Communication (HPCC), pp. 375–380. IEEE Computer Society (June 2012)
20. Wolfe, M., et al.: CUDA Fortran Programming Guide and Reference. The Portland Group, Release (2012)
21. Yan, Y., Grossman, M., Sarkar, V.: JCuda: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 887–899. Springer, Heidelberg (2009)
22. Zaremba, W., Lin, Y., Grover, V.: Jabee: Framework for Object-Oriented Java Bytecode Compilation and Execution on Graphics Processor Units. In: Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, pp. 74–83. ACM (2012)
23. Zhang, W.: Branch-and-Bound Search Algorithms and Their Computational Complexity. Technical report, DTIC Document (1996)

Real-World Loops Are Easy to Predict

Raphael Ernani Rodrigues

Department of Computer Science, UFMG, Brazil
raphael@dcc.ufmg.br

Abstract. The Trip Count of a loop determines how many iterations this loop performs. Several compiler optimizations yield greater benefits for large trip counts, and are either innocuous or detrimental for small ones. However, predicting exactly the trip count of a loop is an undecidable problem in general. Thus, such problem is usually approached through heuristics, which tend to be computationally expensive. In this paper we argue that in most cases there is no need to resort to expensive methods and that in many cases the trip count prediction does not need to be sound. In that sense, we propose a lightweight trip count prediction heuristic. Our method identifies the pattern on which the induction variables of each loop are updated between two iterations and generate symbolic expressions that represent the trip counts of the loops. Such expressions can be evaluated at runtime with O(1) complexity and allow blocks of code to be conditionally executed, depending on the expected trip count. We argue that such technique is useful for speculative optimizations, very common in the world of just-in-time compilers. For instance, if we predict that a loop will iterate for a long time, we can perform more aggressive JIT optimizations. Furthermore, we show that despite the simplicity of our technique, we have accurately predicted nearly 90% of all the interval loops found in millions of lines of C code. The interval loops represent approximately 67% of the total number of loops of the programs.

1 Introduction

Loops represent most of the execution time of a program. For that reason, there is a well-known aphorism that says that "all the gold lays in the loops". Thus, compiler optimizations made inside loops have their benefits multiplied by the number of iterations actually executed. As a consequence of that fact, there is a vast number of works in the literature that are specialized in loop optimizations [19,11,14].

Some optimizations, however, are highly sensitive to the number of iterations of a given loop. For instance, if a given loop iterates a few times in an interpreter, an aggressive optimization made by a Just-In-Time (JIT) compiler may not even pay for the compilation overhead. On other hand, if the same loop iterates thousands of times, the JIT compilation might use more expensive techniques and still have a better end-to-end performance. The number of iterations a loop actually executes is called *Trip Count*. Here we use the same concept of trip count as described by Wolfe *et al.* [19, pp.200].

F.M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 124–138, 2014.

However, in most cases this number is only known at runtime. Rice [16] has demonstrated that statically predicting this information is an undecidable problem. Therefore, this problem can only be partially solved by heuristics. There is a large number of works in the literature that propose different techniques to solve this problem [18,1,9]. The main disadvantage of them is the high computational cost to predict loops. That characteristic makes these solutions impractical to be applied to JIT compilers, for instance.

In this particular work, we argue that, although predicting exactly the trip count of a loop is undecidable, most of the time there is no need to use computationally expensive state-of-the-art methods to compute (an approximation of) it. We propose a heuristic that extracts patterns of the updates of variables' values and estimates the trip count of loops with symbolic expressions. Those expressions might, then, be evaluated at runtime with O(1) complexity. They allow the program to decide dynamically which piece of code to execute, depending on the actual expected number of iterations.

We support our position with a series of experiments. We have analyzed millions of lines of C code, with thousands of different loops present on well-known benchmarks. Our experiments demonstrate that most of the loops are easy to predict and do not demand expensive techniques to be accurately analyzed. We have collected statistics about the structure of thousands of loops that support our claim. According to our research, 70% of the loops have a very simple and well-behaved structure. We have instrumented those loops and we have exactly predicted the trip counts of 90% of them.

The rest of this paper is organized as follows: Section 2 gives us the basis upon which we develop our technique. Sections 3 and 4 describe our algorithms and explain our engineering choices. We experimentally evaluate the performance of our work in Section 5. In Section 6 we discuss how our work is related with existing efforts in the literature. Finally, in Section 7 we discuss possible future directions of this research and make final remarks.

2 Background

Our analysis combines information contained in the Control Flow Graph (CFG) and in the Data Dependence Graph of the program. From the CFG we can extract information about the structure of the analyzed program, like the points of the program where the loops start and stop, and which variables and instructions directly affect the control flow. From the dependence graph we can extract information about the way that the information flows among the variables. Those information allow us to generate symbolic expressions that estimate the trip count of the loops of the program.

The dependence graph [5] is defined in the following way: for each program variable v, we create a node n_v, and for each instruction i in the program we create a node n_i. For each instruction $i : \ v \ = \ f(\ldots, \ u, \ \ldots)$ that defines a variable v and uses a variable u we create two edges: $n_u \rightarrow n_i$ and $n_i \rightarrow n_v$.

Many variables of a program do not affect the predicates that represent the loops' stop conditions. Thus, we do not consider those variables in our analysis,

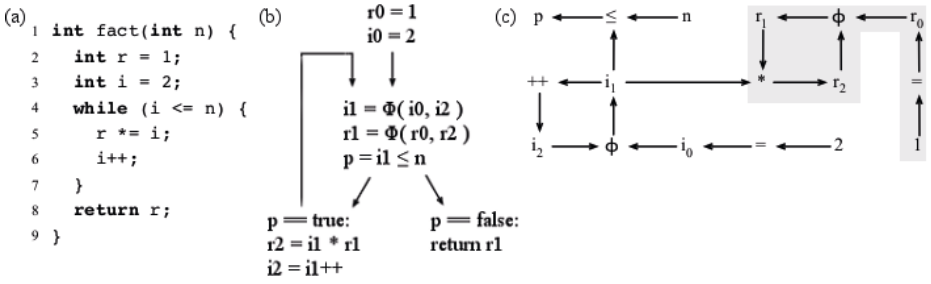


Fig. 1. (a)Example program. (b) CFG of the program, after conversion to SSA form. (c)Dependence graph highlighting nodes that do not affect the loop predicate, after converting the original program into SSA.

because they do not have any impact on the number of iterations of those loops. Therefore, despite of working with a slice of the program that eliminates those instructions, the result of our analysis remains the same. Figure 1 shows a dependence graph for the factorial function and highlights the variables that we can prune before doing our analysis. In this work we use the SSA representation form [3]. We chose this representation form because it is able to give extra precision to our analyses in cases when the same variable is redefined in two different loops.

Natural Loops

According to Appel and Palsberg [2, p.376], a natural loop is a set of nodes S of the control flow graph (CFG) of a program, including a header node H, with the following properties:

- from any node in S there is a path that reaches H;
- there is a path from H to any node that belongs to S;
- any path from a node outside S to a node inside S contains H.

A node PH of the CFG is a pre-header of a natural loop if and only if PH has H as an immediate successor. In this work we normalize the CFG in such a way that every natural loop has one unique pre-header PH, that is executed immediately before the first iteration of the loop. Such normalization gives us a basic block that immediately dominates the loop. This basic block is used by our profiler to initialize the variables that we use to observe the behavior of the loop.

In addition, the *stop condition* of a loop is a boolean expression $E = f(e_1, e_2, \ldots, e_n)$, where each $e_j, 1 \leq j \leq n$ is a value that contributes to the computation of E. Depending on the stop condition we classify the loop into one of the following categories:

- **Interval Loops** - The *stop condition* is an integer comparison instruction that receives two operands e_1 and e_2 and compares them with an inequality ($<$, $\leq$, $>$, or $\geq$).

- **Equality Loops** - The *stop condition* is an integer comparison instruction that receives two operands e_1 and e_2 and compares them with an equality (== or ! =).
- **Other Loops** - Any natural loop that neither is an *Interval Loop* nor is an *Equality Loop*.

Strongly Connected Components
Variables that are redefined during the execution of a loop of the program belong to cycles in the dependence graph. Cycles of redefinitions of variables can be identified by computing the Strongly Connected Components (SCCs) on the dependence graph [17]. The SCCs help us to group the different nodes of the graph that belong to the same redefinition sequence.

After we have computed the SCCs of the dependence graph, we can classify them in the following way:

- **Single-node SCC** - SCCs composed by only one node.
- **Multi-node SCC** - SCCs composed by more than one node. SCCs of this class represent cycles in the dependence graph.

Multi-node SCCs can be divided in two categories:

- **Single-path SCC** - From any node in the SCC there is only one path that starts and ends in the same node and passes through the edges of the SCC at most once.
- **Multi-path SCC** - There is at least one node in the SCC for which there are two or more paths that start and end in the same node and pass through the edges of the SCC at most once.

Single-path SCCs are unconditional sequences of redefinitions of a variable. This pattern of SCC is the easiest to analyze, because it is possible to infer statically what is the effect of one iteration of the loop on the variables of the SCC.

Multi-path SCCs are conditional sequences of redefinitions of a variable. This means that there are conditional branches inside the CFG loop. The amount of branches makes the total number of possible paths to grow exponentially. Thus, this class of SCCs is harder to analyze. Multi-path SCCs can be further classified into two different categories:

- **Single-loop SCC** - The SCC has branches that does not constitute nested loops.
- **Nested-loop SCC** - There are inner loops inside the SCC. In order to avoid non-termination problems, we do not analyze this category of SCC.

Sequences of Redefinitions of Variables
A sequence of redefinitions (SR) is a path in the SCC that starts and ends in the same node and does not repeat any edge. By construction, our dependence graph does not admit self loops, so SRs are always extracted from Multi-node SCCs. A SR can be interpreted to generate the effect of one iteration of the

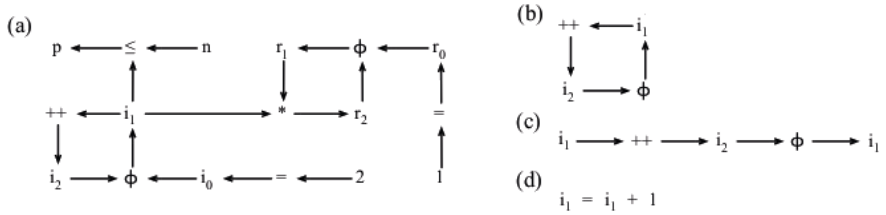


Fig. 2. (a)Dependence graph. (b)Multi-node SCC of the variable i_1. (c)Sequence of redefinitions of the variable i_1. (d)Effect of one iteration on the variable i_1.

program on a given variable. Figure 2 shows an example of SR for one induction variable.

Considering infinite precision, a SR has one of the following classifications:

- **Constant** - after one iteration of the SR, the value of the variable remains the same.
- **Increasing** - after one iteration, the value of the variable is always larger than the initial value.
- **Decreasing** - after one iteration, the value of the variable is always smaller than the initial value.
- **Possibly Oscillating** - after one iteration, we are not able to prove neither an increasing nor a decreasing behavior.

SRs that are classified as *Possibly Oscillating* are placed in this category because at least one of the following reasons is true:

- There is a call instruction in the SR. Currently our analysis is not able to analyze interprocedural SRs.
- There is an operation in the SR that receives an operand X, where the range analysis of X gives a positive upper bound and a negative lower bound.
- The SR depends on SCCs that have been classified as *Possibly Oscillating*.

We can classify a Multi-node SCC within the same categories of a single SR. For that, the classification of all the SRs of the SCC must be combined using a meet operation in the lattice shown in Figure 3. Therefore, the classification of a SCC is the least upper bound of the classifications of the SRs that the SCC contains.

Vectors

In order to achieve good precision without sacrificing efficiency, we propose an abstraction called vectors to predict trip count. We place each numeric variable v of the analyzed program on the real number line, in the point corresponding to the value stored by v. Whenever the value that v stores is changed, we move v to another point of the real line, corresponding to its new value. By doing that, we have observed that some variables have a well-defined behavior along loop iterations, that we can translate into patterns of movement. The vectors are,

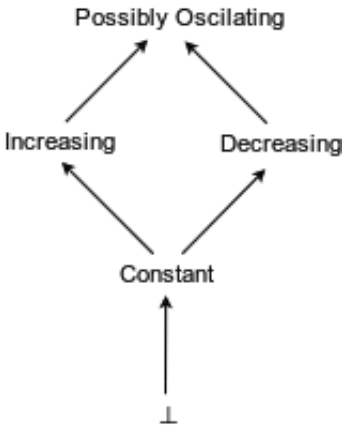


Fig. 3. Lattice of SR classifications

then, the structures that help us to understand those patterns of movement. We borrow the concept from linear physics vectors (magnitude, direction).

A vector is the step given by a variable v after one complete iteration through a SR p. Before the execution of p, we have v_0 stored in v. After the execution of p, v will be redefined with a new value, v_p. We can understand this redefinition of v as a move on the real number line. The step given by v in the line is $v_p - v_0$. Thus, a vector of a variable v extracted from a SR p is $\Delta v_p = v_p - v_0$.

The sign of Δv_p indicates the direction of the vector (i.e. the direction to where we are moving v). Vectors may be defined by symbolic expressions involving other variables of the program. This characteristic generates a chain of dependencies that brings the need to process the SCCs in topological order. If the SCC of a variable n is classified as *Possibly Oscillating*, then the vectors that depend on n are unknown.

Patterns of Movement

When variables have their values updated always by vectors with the same characteristics, some patterns of movement are noticeable:

- **Stationary** - variables updated by vectors with length equal to zero.
- **Constant Speed** - variables updated by vectors with constant length. In this case, on each iteration, the variable is moved a constant distance from its previous location, creating a linear behavior.
- **Constant Acceleration** - variables updated by a vector that has a linearly increasing length. This kind of vector is generated by a linear expression involving a *Constant Speed* variable, creating a quadratic behavior.
- **Constantly Increasing Acceleration** - variables updated by a vector that is generated by a linear expression involving a *Constant Acceleration* variable, creating a cubic behavior.

- **More Than Cubic** - variables updated by a vector that is generated by a linear expression involving a *Constantly Increasing Acceleration* or *More Than Cubic* variable, creating a more-than-cubic behavior.
- **Unknown** - Occurs when:
  - Variables are updated with vectors that depend on variables with *Unknown* movement patterns.
  - Variables are updated with vectors that have their length decreased on each iteration.

3 A Trip Count Algorithm Based on Vectors

Most of the loops have their number of iterations controlled by data that come from the input, so a purely static analysis is not precise enough to solve this problem. We propose, then, a hybrid solution involving a static and a dynamic step: we statically analyze the program and generate symbolic expressions that represent the estimated trip counts of its loops. Dynamically, during its execution, the instrumented program evaluates those expressions with O(1) complexity. Other optimizations might use the result of those expressions to make decisions at runtime, depending on the expected trip count. At the end of the compilation process, unused expressions can be trivially removed by a dead code elimination procedure.

Algorithm 1. Trip Count Instrumentation Based on Vectors

Input: Program P
Output: Program P with new instructions that estimate the maximum trip count of the loops

```
1: for all Loop l ∈ P do
2:     if isIntervalLoop(l) or isEqualLoop(l) then
3:         Variable op1 = getFirstOperand(l.getStopCondition())
4:         Variable op2 = getSecondOperand(l.getStopCondition())
5:         Expression step = estimateMinimumStep(op1, op2)
6:         if ∃ step then
7:             Insert instructions that compute the expression |op1 − op2|/step before the first iteration of l.
8:         end if
9:     end if
10: end for
```

Algorithm 1 presents the static analysis needed to generate the trip count expressions using vectors. Our heuristic only covers *Interval Loops* and *Equality Loops*. Once we have collected both operands op_1 and op_2 of the stop condition of a loop l, we have to estimate the step of approximation of the two variables in the real numbers line. In order to estimate the trip count of a loop, we must be able to evaluate op_1 and op_2 before the first iteration. Thus, both operands must be integer expressions that do not produce side effects when evaluated. Finally, if there is a well defined behavior of update of both operands, then $estimateMinimumStep(op_1,\ op_2)$ will return a valid step and we can estimate the trip count. Otherwise, we are not able to estimate the trip count of l.

Algorithm 2. estimateMinimumStep: Estimate the minimum step of approximation of variables in the real line.

Input: Pair of Variables $op_1,\ op_2$
Output: Expression *step* with the minimum step.

```
1: Vector v1 = getMinVector(op1)
2: Vector v2 = getMinVector(op2)
3: if ∃ v1 and ∃ v2 then
4:     return |v1 − v2|
5: else
6:     return null
7: end if
```

Algorithm 2 shows how we estimate the minimum step given the minimum vectors of the two variables that control the stop condition of the loop. The minimum step leads to the maximum trip count. Minor adaptations are required to generate the maximum step and, finally, the minimum trip count. The variables will only have a minimum vector if they have a monotonic behavior i.e. whenever the variables move in the real line, they move in the same direction.

Algorithm 3. getMinVector: Generate the minimum vector of a given monotonic variable

Input: Variable v
Output: Vector $\vec{V}$ with the minimum length.

```
1: Vector V⃗ = ⊥
2: for all RedefinitionSequence rs of v do
3:     Vector Tmp⃗ = evaluateDelta(rs)
4:     V⃗ = joinVectors_min(V⃗, Tmp⃗)
5:     if V⃗ == unknown then
6:         break
7:     end if
8: end for
9: return V⃗
```

Algorithm 3 generates the minimum vector for a given variable. In order to generate such vector, we have to symbolically evaluate every *Sequence of Redefinition* of that variable and join the results in a vector. The join operation has two steps: first we check the direction of both vectors. If the vectors have opposite directions or one of the vectors is *unknown*, the result of $joinVectors_min(\vec{V},\ \overrightarrow{Tmp})$ is *unknown*. Otherwise, we take the vector with the minimum length as the result of the join.

4 A Simplified Trip Count Algorithm Based on Vectors for JIT Compilers

With the massive increase of the usage of the World-Wide-Web and the introduction of many new architectures that must run the same programs, it is essential to have portable programs. Code interpreting provides easy portability of programs, because just the interpreter must be translated into the different

architectures, instead of any program of a given language. However, code interpreting is slow and excessively consumes resources. In this context, Just-In-Time (JIT) compilers are used to overcome the inefficiencies that code interpreters inherently have [15].

JIT compilers work by compiling pieces of code right before they are executed. Whenever the controller thread tries to execute some function that has no native code available, the controller thread calls the compiler before executing the function. That means that the execution stops while the JIT compiler is generating native code. Therefore, the JIT compiler must generate the most optimized possible code in the minimum time, because the total time (compiling + execution) must be lower than the interpreting time, otherwise there is no point in compiling those programs. Because of that, JIT compilers must use extremely lightweight algorithms to keep the compiling time as low as possible.

Here we present a simplification in our trip count prediction heuristic in order to be able to apply it in JIT compilers. As we have observed in Section 5, 90% of the natural loops are either Interval Loops or Equality Loops. Moreover, most of our vectors are constant speed vectors with length equal to one. From those facts, in the simple heuristic we assume that in every Interval or Equality loops the minimum step of approximation of op_1 and op_2 is equal to one. Thus, the estimated trip count is $|op_2 - op_1|$ and we avoid calling $EstimateMinimumStep(op_1,\ op_2)$. Our heuristic generates the expression that estimates the trip count with $O(1)$ complexity. The complexity of the analysis of the whole program is $O(n)$, where n is the number of natural loops of the program.

5 Experimental Results

We have implemented a prototype of our analyses in the LLVM compiler, version 3.3. We have analyzed more than 500 programs, including the benchmarks of the LLVM test-suite and the benchmarks of SPEC 2006 CPU. In this section we will focus the discussion in the results obtained with the analysis of the benchmarks of SPEC 2006 CPU.

Most of the loops of the programs have a simple structure, and that means that the analysis does not need to be complicated in order to cover almost all loops of a program. Table 1 analyzes the structure of natural loops of programs. According to Ferrante [5], natural loops are *single-entry* regions. We have observed that 65.92% of the loops have just one stop instruction, so they are *single-entry* and *single-exit* regions. However, 39.87% of the loops are nested inside other loops. Those numbers tell us that despite of the simplicity of most loops, a considerable amount of them is nested, so loop analyses that does not support nested loops leave a large number of loops uncovered.

We have also identified a pattern in the stop conditions of the loops. Table 2 shows that approximately 85% of the natural loops have a single integer comparison as the stop condition. Moreover, the vast majority of those loops are interval loops, the easiest kind of loop to analyze. We have also observed similar proportions while analyzing the rest of our benchmarks. Those numbers are

Table 1. Natural Loops in the Control Flow Graph. L: number of natural loops. NL: number of nested loops. SEL: number of loops that have a single exit point.

Program	L	NL	% NL/L	SEL	% SEL/L
433.milc	426	211	49.53%	399	93.66%
444.namd	623	418	67.09%	593	95.18%
447.dealII	6526	2695	41.30%	3412	52.28%
450.soplex	742	181	24.39%	554	74.66%
470.lbm	23	10	43.48%	23	100.00%
401.bzip2	238	85	35.71%	150	63.03%
403.gcc	4614	1357	29.41%	3202	69.40%
429.mcf	50	9	18.00%	39	78.00%
445.gobmk	1288	482	37.42%	913	70.89%
456.hmmer	881	245	27.81%	740	84.00%
458.sjeng	267	62	23.22%	201	75.28%
462.libquantum	98	13	13.27%	90	91.84%
464.h264ref	1870	1008	53.90%	1784	95.40%
471.omnetpp	465	66	14.19%	249	53.55%
473.astar	119	37	31.09%	104	87.39%
483.xalancbmk	3106	259	8.34%	1611	51.87%
Total	21336	7138	33.46%	14064	65.92%

Table 2. Classification of Natural Loops according to their stop conditions. L: number of natural loops. IL: number of *Interval Loops*. EL: number of *Equality Loops*. OL: number of *Other Loops*.

Program	L	IL	% IL/L	EL	% EL/L	OL	% OL/L
433.milc	426	417	97.89%	5	1.17%	4	0.94%
444.namd	623	494	79.29%	7	1.12%	122	19.58%
447.dealII	6526	4597	70.44%	604	9.26%	1325	20.30%
450.soplex	742	572	77.09%	101	13.61%	69	9.30%
470.lbm	23	23	100.00%	0	0.00%	0	0.00%
401.bzip2	238	201	84.45%	29	12.18%	8	3.36%
403.gcc	4614	2103	45.58%	1954	42.35%	557	12.07%
429.mcf	50	17	34.00%	28	56.00%	5	10.00%
445.gobmk	1288	1098	85.25%	131	10.17%	59	4.58%
456.hmmer	881	697	79.11%	109	12.37%	75	8.51%
458.sjeng	267	117	43.82%	128	47.94%	22	8.24%
462.libquantum	98	88	89.80%	6	6.12%	4	4.08%
464.h264ref	1870	1789	95.67%	19	1.02%	62	3.32%
471.omnetpp	465	283	60.86%	82	17.63%	100	21.51%
473.astar	119	108	90.76%	1	0.84%	10	8.40%
483.xalancbmk	3106	1687	54.31%	752	24.21%	667	21.47%
Total	21336	14291	66.98%	3956	18.54%	3089	14.48%

favorable to our heuristics, because we take advantage of the simplicity of the loops to produce precise results with simple algorithms.

Table 3 shows the statistics collected while analyzing the dependence graphs of the programs. 85.40% of the Multi-Node SCCs are Single-Path SCCs. That means that there is only one SR for the variables of such SCCs. Moreover, just 7.70% of the Multi-Node SCCs have nested cycles and do not fulfill the requirements of our analysis. All the presented data confirms that the programs have a structure that is suitable for our heuristics to produce accurate results.

In order to estimate the trip count of a loop, our prototype must be able to infer the values that Op_1 and Op_2 store before the first iteration. Op_1 and Op_2 are the operands of the stop condition of the loop. This information is not always

Table 3. Classification of Strongly Connected Components in the Dependence Graph. SN: number of *Single-Node* SCCs. MN: number of *Multi-Node* SCCs. SP: number of *Single-Path* SCCs. MP: number of *Multi-Path* SCCs. SL: number of *Single-Loop* SCCs. NL: number of *Nested-Loop* SCCs.

Program	SN	MN	SP	% SP/MN	MP	SL	% SL/MP	NL	% NL/MN
433.milc	2507	426	409	96.01%	17	11	64.71%	6	1.41%
444.namd	5879	781	604	77.34%	177	6	3.39%	171	21.90%
447.dealII	79169	7249	6077	83.83%	1172	505	43.09%	667	9.20%
450.soplex	13032	807	683	84.63%	124	53	42.74%	71	8.80%
470.lbm	94	24	23	95.83%	1	1	100.00%	0	0.00%
401.bzip2	3610	214	171	79.91%	43	16	37.21%	27	12.62%
403.gcc	123775	5121	4513	88.13%	608	276	45.39%	332	6.48%
429.mcf	1022	54	40	74.07%	14	7	50.00%	7	12.96%
445.gobmk	17675	1555	1283	82.51%	272	163	59.93%	109	7.01%
456.hmmer	12215	946	825	87.21%	121	67	55.37%	54	5.71%
458.sjeng	3337	276	221	80.07%	55	38	69.09%	17	6.16%
462.libquantum	1439	123	100	81.30%	23	8	34.78%	15	12.20%
464.h264ref	21502	1946	1841	94.60%	105	27	25.71%	78	4.01%
471.omnetpp	12383	470	379	80.64%	91	39	42.86%	52	11.06%
473.astar	2591	138	118	85.51%	20	7	35.00%	13	9.42%
483.xalancbmk	57181	3024	2486	82.21%	538	373	69.33%	165	5.46%
Total	357411	23154	19773	85.40%	3381	1597	47.23%	1784	7.70%

Table 4. Trip Count Instrumentation. IL: interval loops. IIL: instrumented interval loops. EL: equality loops. IEL: instrumented equality loops.

Program	# IL	# IIL	% IIL/IL	# EL	# IEL	% IEL/EL
433.milc	417	391	93.76%	5	3	60.00%
444.namd	494	469	94.94%	7	1	14.29%
447.dealII	4597	3535	76.90%	604	77	12.75%
450.soplex	572	422	73.78%	101	48	47.52%
470.lbm	23	23	100.00%	0	0	-
401.bzip2	201	186	92.54%	29	7	24.14%
403.gcc	2103	1798	85.50%	1954	192	9.83%
429.mcf	17	7	41.18%	28	1	3.57%
445.gobmk	1098	1040	94.72%	131	56	42.75%
456.hmmer	697	664	95.27%	109	39	35.78%
458.sjeng	117	106	90.60%	128	17	13.28%
462.libquantum	88	77	87.50%	6	1	16.67%
464.h264ref	1789	1411	78.87%	19	8	42.11%
471.omnetpp	283	238	84.10%	82	29	35.37%
473.astar	108	80	74.07%	1	1	100.00%
483.xalancbmk	1687	1403	83.17%	752	108	14.36%
Total	14291	11850	82.92%	3956	588	14.86%

possible to be inferred, because sometimes one of the operands is the result of a call to other function. In cases like this, we do not know the value of the operand before the loop starts and, consequently, we are not able to estimate its trip count. Thus, both operands must be integer expressions that do not produce side effects when evaluated. Table 4 shows the number of loops of which we are able to infer the trip count. For instance, we were able to estimate the trip count of 85.50% of the interval loops of the benchmark 403.gcc, while we were able to instrument just 9.83% of its equality loops. We have investigated this and we observed that most of the 403.gcc's equality loops are bounded by comparisons

between pointers. The same was observed in other programs. Because of that, we should focus only in the interval loops.

We have developed a profiler that collects the estimated trip count and the real trip count during an actual execution of the benchmarks. The result of our profiler lets us to observe how accurate are our heuristics. We have split our accuracy results into seven categories according to the actual number N of iterations:

- $[0, \sqrt{N}]$: Occurs when the estimated trip count is less or equal the square root of the actual trip count. For example, if we estimate that a loop will iterate 2 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $]\sqrt{N}, N/2]$: Occurs when the estim ted trip count is greater than the square root of the actual trip count but is less or equal its half. For example, if we estimate that a loop will iterate 4 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $]N/2, N[$: Occurs when the estimated trip count is greater than the half of the actual trip count but is less than the trip count. For example, if we estimate that a loop will iterate 8 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $[N, N]$: Occurs when the estimated trip count equals the actual trip count. For example, if we estimate that a loop will iterate 10 times and it iterates 10 times during its execution, this loop will fall into this category.
- $]N, 2 * N]$: Occurs when the estimated trip count is greater than the actual trip count, but is less or equal to two times the actual trip count. For example, if we estimate that a loop will iterate 16 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $]2 * N, N^2]$: Occurs when the estimated trip count is greater than two times the actual trip count, but is less or equal to the power of two of the actual trip count. For example, if we estimate that a loop will iterate 32 times and it iterates 10 times during its execution, this loop will be classified into this category.
- $]N^2, +\infty]$: Occurs when the estimated trip count is greater than the power of two of the actual trip count. For example, if we estimate that a loop will iterate 128 times and it iterates 10 times during its execution, this loop will be classified into this category.

Table 5 shows the comparison between the estimated trip count and the actual trip count that we have collected with our profiler. The subtotal lines contain only the SPEC CPU benchmarks, while the total lines also include more than 300 benchmarks distributed with LLVM. While running the programs, each time a loop stops, we collect the actual trip count and compare it with the estimated trip count. Thus, the numbers that we presented is the number of *instances* of loops, instead of the number of natural loops. We did this because we may predict correctly the trip count for some instances and may predict wrongly for other instances of the same CFG loop. Table 6 shows information about the programs

Table 5. Trip Count Profiler - Trip count estimated using vectors

Program	$[0, \sqrt{N}]$	$]\sqrt{N}, N/2]$	$]N/2, N[$	$[N, N]$	$]N, 2*N]$	$]2*N, N^2]$	$]N^2, +\infty]$
milc	14	0	0	435,514,912	38,360	9,984	1,032,930
namd	0	0	0	21,602,695	8,064	3,168	0
soplex	1,851	367	122	186,943	12,782	10,219	43,338
lbm	0	0	0	53,397	0	0	0
bzip2	8,616,650	2	311,724	13,204,855	14,195,603	1,128,948	28,939,274
gcc	433,588	17	326	17,240,735	1,851,284	278,164	336,422
mcf	96,576	87	42	555	2,643,736	634,623	1,705,369
gobmk	8,392	20	400	651,081	70,492	117	20,141
hmmer	0	0	0	31,551,408	8,512,797	3,893,744	3,273,429
sjeng	0	620	2,565,378	41,787,788	3,423,766	7,917	1,038,075
libquantum	0	0	0	8,182,095	0	1	0
h264ref	6,749,850	0	0	311,274,945	13,427,840	57,300	228,711
astar	7,147	0	0	74,614,711	602,244	2,550	609,708
Subtotal	15,914,068	1,113	2,877,992	955,866,120	44,786,968	6,026,735	37,227,397
Subtotal (%)	1.50%	0.00%	0.27%	89.95%	4.21%	0.57%	3.50%
Total	25,525,142	2,078	2,922,080	4,134,074,825	163,974,403	11,363,892	400,209,181
Total (%)	0.54%	0.00%	0.06%	87.25%	3.46%	0.24%	8.45%

Table 6. Trip Count Profiler - Trip count estimated using simplified heuristic

Program	$[0, \sqrt{N}]$	$]\sqrt{N}, N/2]$	$]N/2, N[$	$[N, N]$	$]N, 2*N]$	$]2*N, N^2]$	$]N^2, +\infty]$
milc	14	0	0	435,514,912	38,360	9,984	1,032,930
namd	0	0	0	21,602,688	8,065	3,174	0
soplex	1,851	367	112	186,939	12,784	10,231	43,338
lbm	0	0	0	53,333	0	64	0
bzip2	5,270,006	2	311,724	14,386,219	15,987,072	1,502,759	28,939,274
gcc	420,390	17	326	17,252,944	1,841,701	283,373	343,054
mcf	96,576	87	42	555	2,643,736	634,623	1,705,369
gobmk	8,392	20	400	651,081	70,492	117	20,141
hmmer	0	0	0	31,551,408	8,512,797	3,893,744	3,273,429
sjeng	0	620	2,565,378	41,787,788	3,423,766	7,917	1,038,075
libquantum	0	0	0	8,182,095	0	1	0
h264ref	367,010	0	0	302,394,768	12,636,622	5,636,387	10,703,859
astar	7,147	0	0	74,614,711	602,244	2,550	609,708
Subtotal	6,171,386	1,113	2,877,982	948,179,441	45,777,639	11,984,924	47,709,177
Subtotal (%)	0.58%	0.00%	0.27%	89.22%	4.31%	1.13%	4.49%
Total	10,762,387	2,094	2,882,136	3,996,856,652	227,506,781	53,384,130	441,012,280
Total (%)	0.23%	0.00%	0.06%	84.46%	4.81%	1.13%	9.32%

that had their trip counts estimated using the simplified heuristic, following the same rules used to build table 5.

By analyzing table 5 we can observe that our heuristic is very precise. 87.25% of the trip counts that we have predicted were the same as the actual trip count. Furthermore, we have observed that more than 99% of the estimated trip counts were equal or greater than the actual trip counts. When we analyze the results obtained with the simplified heuristic, we also find some impressive numbers. As expected, the vector heuristic has better results than the simplified heuristic, but the difference was small. Our predictions were exact in 84.46% of the cases, despite of the extreme simplicity of the algorithm. We also have observed the same over-approximation that we have noticed with the complete vector heuristic. However, it is important to keep in mind that we are not able to instrument 100% of the loops of the programs. In this experiment we only consider Interval Loops, that account for 67% of them.

6 Related Works

It is possible to estimate the trip count of loops in many different ways, in a trade-off between speed and precision. In order to estimate the trip count of loops, many authors have used abstract interpretation [4,10,6,8]. Others have used symbolic execution to achieve similar goals [13,12]. Although those techniques are quite powerful, they are also computationally expensive. Thus, their application is limited by the size of programs to be analyzed. Nevertheless, the high complexity does not mean perfect precision. Some of those works have restrictions with regards to the structure of the analyzed loops. For instance, some of them only analyze loops with a single path and are very conservative while analyzing nested loops. Our work aims to find a better balance between speed and precision.

In an effort similar to ours, Gulwani *et al.* [7] have developed a new approach to estimate the number of iterations of a loop. They have proposed the *Control-Flow Refinement*, a conversion of the programs into a suitable representation, that allowed them to handle programs that other algorithms were not able to analyze. That representation allowed them to find symbolic bounds for 90% of the programs they have analyzed. However, they still rely on expensive techniques. For instance, their implementation requires a theorem prover. Such tools often rely on solutions to NP-complete problems. Differently to their work, here we present two heuristics to estimate the number of iterations of loops that use simpler techniques. Despite of the simplicity of our algorithms, our results show that we offer a good precision without resorting to expensive techniques.

7 Conclusion

In this paper we have discussed the prediction of the number of iterations of loops. We have indicated the usefulness of such information to decide at runtime which code to execute based on the estimated trip count of loops. We also have classified the loops into a taxonomy proposed by us, which allowed us to better understand where the most promising optimizing opportunities are. In addition, we proposed the Vectors, an abstraction inspired by physics to represent patterns of updates of variables on the real line. Furthermore, we have proposed two heuristics to estimate the trip count of loops, based on our Vectors. Finally, we have evaluated the precision of our heuristics on some test suites, observing 87.25% of accuracy while analyzing interval loops.

References

1. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010)
2. Appel, A.W., Palsberg, J.: Modern Compiler Implementation in Java, 2nd edn. Cambridge University Press (2002)

3. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Kenneth Zadeck, F.: Efficiently computing static single assignment form and the control dependence graph. TOPLAS 13(4), 451–490 (1991)
4. Ermedahl, A., Gustafsson, J.: Deriving annotations for tight calculation of execution time. In: Lengauer, C., Griebl, M., Gorlatch, S. (eds.) Euro-Par 1997. LNCS, vol. 1300, pp. 1298–1307. Springer, Heidelberg (1997)
5. Ferrante, J., Ottenstein, K., Warren, J.: The program dependence graph and its use in optimization. TOPLAS 9(3), 319–349 (1987)
6. Gulavani, B.S., Gulwani, S.: A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 370–384. Springer, Heidelberg (2008)
7. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: ACM Sigplan Notices, vol. 44, pp. 375–385. ACM (2009)
8. Gulwani, S., Mehra, K.K., Chilimbi, T.: Speed: precise and efficient static estimation of program computational complexity. In: ACM SIGPLAN Notices, vol. 44, pp. 127–139. ACM (2009)
9. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In: 27th IEEE International Real-Time Systems Symposium, RTSS 2006, pp. 57–66. IEEE (2006)
10. Halbwachs, N., Proy, Y.-E., Roumanoff, P.: Verification of real-time systems using linear relation analysis. Formal Methods in System Design 11(2), 157–185 (1997)
11. Kennedy, K., Allen, R.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann (2001)
12. Liu, Y.A., Gomez, G.: Automatic accurate time-bound analysis for high-level languages. In: Müller, F., Bestavros, A. (eds.) LCTES 1998. LNCS, vol. 1474, pp. 31–40. Springer, Heidelberg (1998)
13. Lundqvist, T., Stenström, P.: Integrating path and timing analysis using instruction-level simulation techniques. In: Müller, F., Bestavros, A. (eds.) LCTES 1998. LNCS, vol. 1474, pp. 1–15. Springer, Heidelberg (1998)
14. Park, E., Cavazos, J., Pouchet, L.-N., Bastoul, C., Cohen, A., Sadayappan, P.: Predictive modeling in a polyhedral optimization space. International Journal of Parallel Programming 41(5), 704–750 (2013)
15. Plezbert, M.P., Cytron, R.K.: Does “just in time"=“better late than never"? In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 120–131. ACM (1997)
16. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society 74(2), 358–366 (1953)
17. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. 1(2), 146–160 (1972)
18. Tetzlaff, D., Glesner, S.: Static prediction of loop iteration counts using machine learning to enable hot spot optimizations. In: 2013 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pp. 300–307. IEEE (2013)
19. Wolfe, M.J., Shanklin, C., Ortega, L.: High performance compilers for parallel computing. Addison-Wesley Longman Publishing Co., Inc. (1995)

A Hybrid Framework to Accelerate Adaptive Compilation Systems

Gabriel Krisman Bertazi[1], Anderson Faustino da Silva[2], and Edson Borin[1]

[1] Institute of Computing, University of Campinas, Brazil
gabriel.bertazi@students.ic.unicamp.br, edson@ic.unicamp.br
[2] State University of Maringá, Brazil
anderson@din.uem.br

Abstract. Virtual execution is a method to reduce the prohibitive overhead of the execution step on adaptive compilation systems. Nevertheless it may fail to identify the best compiler optimizations set, reducing the speedup that could be achieved by the adaptive compilation system. We discuss the shortcomings of the virtual execution method and propose a hybrid mechanism, which leverages the virtual execution method to select a few optimizations sets before performing the execution step to select the best set of optimizations.

Keywords: Code optimization and adaptive compilation systems.

1 Introduction

Modern compilers employ several algorithms to optimize the code during the compilation process. Nevertheless, some optimizations might not be well suited for every application, and there are several cases where applying an optimization may even decrease the application's performance. Even if a specific optimization does not deteriorate performance by itself, its interaction with other optimizations being applied to the same code might cause it to impact the code's performance in a bad way [3]. An approach to reduce the bad impact a fixed set of optimizations might have on the quality of the compiled code, is to search, at compilation time, for a subset of optimizations that maximizes performance for a particular source code. This process is known as adaptive compilation.

One of the biggest drawbacks of current adaptive compilation systems is the execution of the compiled code. The execution is usually employed to measure the performance of the code generated by each optimization set. However, the amount of time a program takes to execute is usually hard to predict and may be arbitrarily long because it is not directly bound to the size of the code. Notice that, even a small application with a tight loop may take a long time to finish the execution process.

The virtual execution method, proposed by Cooper *et al.* [2], aims at accelerating adaptive compilation systems by predicting the performance of compiled code via static analysis, instead of code execution. In this work we study the

F.M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 139–144, 2014.

impact of the virtual execution method on the quality of the selected optimizations sets and show that this method may not always achieve the best results when compared to the traditional adaptive compilation systems.

We also propose a hybrid mechanism that uses virtual execution to perform the heavy work of selecting a short list of optimizations sets, which we show empirically is very likely to contain a good set of optimizations, and then apply a refinement step, in which we actually execute the generated code to select the best set of optimizations for the application. Our results indicate that this approach achieves better results than the virtual execution framework on several SPEC CPU 2006 applications.

The rest of this paper is organized as follows: Section 2 presents a brief overview of adaptive compilation and the virtual execution method; Section 3 discuss our hybrid framework; Section 4 presents our experimental results; Finally, Section 5 presents the conclusions and future work.

2 Adaptive Compilation and Virtual Execution

The search for an optimal subset of optimizations is usually an iterative process, which involves generating a set of optimizations, compiling the source code with this set of optimizations, executing the program to verify its performance and deciding whether the optimizations set is the best one for the application.

The execution phase, used to measure the quality of each optimizations set, can take a large amount of time to finish, even for small programs. Associated with the high cost of this phase, the high number of possible optimization subsets is a major reason for the long time required to compile a program with classical adaptive compilation methods.

Cooper *et al.* [2] proposed the Virtual Execution method as an attempt to address the overhead caused by the executions of the code generated by the optimizations sets. It reduces the execution overhead by replacing the execution step with an static estimation of the code's performance, using the frequency of basic blocks. In this approach, the code is executed only once, without optimizations, to collect the basic blocks execution frequencies. This information is then propagated through the compiler optimizations phases and used later to estimate the number of instructions that would be executed by the optimized code. This number is then used as an indicative of the performance of the code, enabling the system to select the best set of optimizations without actually executing the optimized the code.

As we show later, the code that executes the least amount of instructions is not always the fastest one. Hence, the virtual execution method may still select a poor optimizations set.

3 The Hybrid Framework

The hybrid framework is a two-phase process. In the first phase, it leverages the virtual execution method to search for the `K` best optimizations sets using the

instruction count criteria. Then, in the second phase, it performs a refinement step, where it actually executes the code compiled with each of the K sets, and compares the execution time to identify the one that provides the best speedup.

The first phase of the algorithm behaves in a similar way as described in previous works on adaptive compilation systems [1,2,3,5]. In our experiments we use an implementation of the Case-based reasoning algorithm presented by Lima et al [3] to find sets of optimizations that could provide us with good speedups. Each optimization set is passed to the compiler, which applies them in turn and estimates its performance using the virtual execution mechanism. The result of this phase is a list of K optimizations sets annotated with their respective performance, estimated by the virtual execution mechanism.

Our results indicate that, for most benchmarks, a small list with the best optimizations sets ($K \leq 5$) selected accordingly to the virtual execution performance criteria is likely to include at least one good set of optimizations for the application. In this sense, the second phase consists on the execution of the code produced by the K sets to select the best performing one. Since this phase executes only K binaries, the overall overhead associated with the execution phase is significantly reduced.

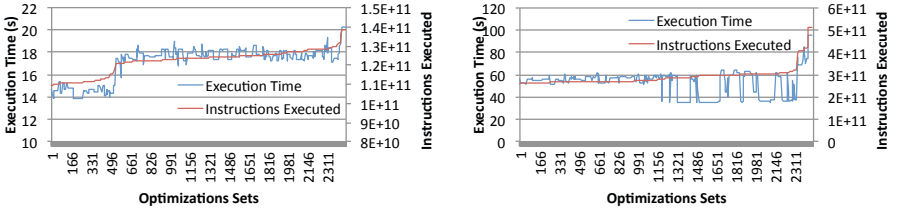
4 Experimental Results

In our experiments, we employed the Case-based Reasoning (CBR) algorithm [3] to search for the best sets of LLVM compiler optimizations to compile the SPEC CPU 2006 [4] programs. In order to perform the experiments on a reasonable time, we measured the performance achieved by each optimizations set by running the compiled code using the training input data set. For each optimizations set, we executed the application 10 times and computed the average runtime. We also used the `PerfSuite` tools[1] to compute the average number of instructions executed. The applications were compiled with LLVM 3.4[2] and executed on a Core I7-2600 with 4 GB of RAM running Ubuntu 13.10.

Our experimental results indicate that there is a correlation between the number of instructions executed and the execution time of the code generated for most of the SPEC CPU 2006 applications. As an example, Figure 1a shows the number of instructions executed and the execution time for each one of the executables produced by the 2479 optimizations sets explored by the CBR algorithm when compiling the PERLBENCH benchmark. Notice that, even though there is some variance on the execution time, the number of executed instructions is a good indicator of the execution time.

The HMMER application, on the contrary, exhibited very little correlation between the number of instructions executed and the execution time. As we can see in Figure 1b, the fastest executables are not the ones that execute the least amount of instructions. The results also indicate that one would have to select a large value for K (> 1000) in order to ensure that the hybrid framework selects a good candidate on the first phase.

[1] `http://perfsuite.ncsa.illinois.edu/`, Accessed: 2014-05-22.
[2] `http://www.llvm.org/`, Accessed: 2014-05-22.



(a) 400.Perlbench (b) 456.hmmer

Fig. 1. Execution time and number of instructions executed for perlbench and hmmer applications

The previous results suggest that selecting the executable that executes the least amount of instructions may not be a good approach to find the ones that execute in less time. However, as we discuss below, for most of the SPEC CPU 2006 applications, selecting the best optimizations set using the hybrid framework with `K=1` does not affect the outcome of the process when compared to, the most expensive, real execution.

In our experiment we assume that the optimizations set selected by the compiler flag `-O2` is our baseline. Also, if the framework does not find an optimizations set that is better than the baseline we select the baseline itself. Figure 2 shows the speedups (slowdown for values < 1.0) achieved on top of the baseline when we use the hybrid framework with `K=1`, 5, 11, 62, and 2076. When `K=2076` all the optimizations sets are selected for the "real execution" phase, ensuring the results are equivalent to the traditional adaptive compilation frameworks.

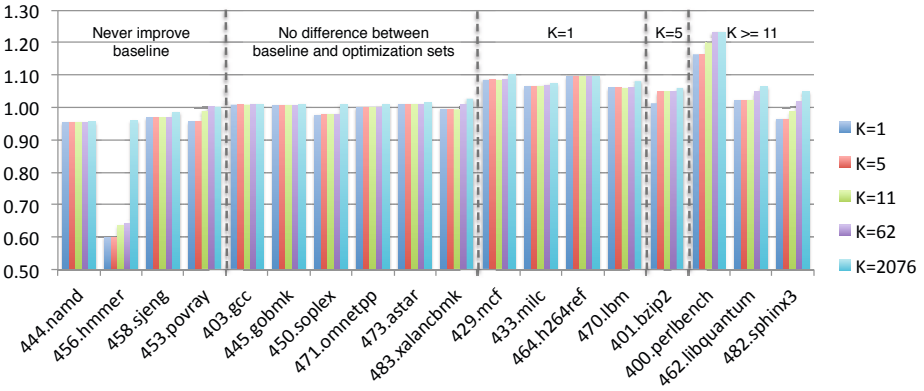


Fig. 2. Speedups achieved by the hybrid framework for `K=1`, 5, 11, 62 and 2076

Notice that for 10 applications neither the traditional (K=2076) nor the hybrid framework (K=1 to 62) is capable of finding an optimizations set that produces better executables than the baseline one. The HMMER benchmark is one of these benchmarks. For this benchmark, the hybrid framework produces

worse results than the traditional one, however, all the results, including the ones achieved by the traditional framework are worse than the baseline. Hence, in both case, the results would be the baseline itself.

The hybrid framework selected good optimizations sets for 4 benchmarks: MCF, MILC, H264REF, LBM. Notice that, even when K=1, the results achieved by the framework are very similar to the ones achieved by a traditional framework. For the BZIP benchmark, the hybrid framework was able to find a good set when K=5. The PERLBENCH benchmark required the hybrid framework to select 11 candidates in the "virtual execution" phase to ensure the "real execution" phase finds a good set of optimizations. The LIBQUANTUM and SPHINX3 required the hybrid framework to select more than 60 candidates in the first phase.

5 Conclusion and Future Work

On this paper we discuss the validity of employing the number of instructions executed as a single measure to predict application's performance on adaptive compilation systems using virtual execution. We show that there usually is a correlation between the number of instructions executed and the program's execution time, even though on most cases this metric alone fails to select the best optimization set found during the optimizations search phase.

We also present a hybrid approach to address the problem of not selecting the best compiler optimizations set found on the virtual execution phase, by expanding the number of test sets to be taken into account, on a latter refinement step. We show that this method improves the quality of the optimizations set selected by the adaptive compilation system, which is reflected on the speedup of several programs of the benchmarks we analyzed.

As a work in progress, the future work should focus on finding new metrics to improve the time estimation performed by the virtual execution system. This would reduce the number of tests performed on the second phase of our framework, reflecting on smaller compilation times and on higher speedups of programs compiled with this method.

Acknowledgments. We thank the referees for their valuable comments. We also thank Petrobras, FAPESP and CNPq for supporting this work.

References

1. Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M.F.P., Temam, O.: Rapidly selecting good compiler optimizations using performance counters. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO 2007, pp. 185–197. IEEE Computer Society, Washington, DC (2007)
2. Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S., Subramanian, D., Torczon, L., Waterman, T.: ACME: Adaptive compilation made efficient. SIGPLAN Not. 40(7), 69–77 (2005)

3. de Lima, E.D., de Souza Xavier, T.C., da Silva, A.F., Ruiz, L.B.: Compiling for performance and power efficiency. In: 2013 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), pp. 142–149 (September 2013)
4. Henning, J.L.: Spec cpu2006 benchmark descriptions. SIGARCH Computer Architecture News 34(4), 1–17 (2006)
5. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), pp. 319–332 (2006)

Transactional Boosting for Haskell

André Rauber Du Bois, Maurício Lima Pilla, and Rodrigo Duarte

Computação - CDTec, UFPel, Pelotas-RS, Brazil
{dubois,pilla,rmduarte}@inf.ufpel.edu.br

Abstract. Transactional boosting is a methodology used to transform highly concurrent linearizable objects into highly concurrent *transactional* objects. In this paper we describe a STM Haskell extension that allows programmers to write *boosted* versions of highly concurrent abstract data types. Although the technique can only be applied to abstract types that have certain properties, when used correctly, we obtain transactional versions of existing types that are much faster than if they were implemented with pure STM Haskell.

1 Introduction

Transactional memory is a higher level alternative to lock based synchronization in concurrent programming. In this model, all accesses to shared memory are grouped as transactions that can execute concurrently and, if no conflicting accesses to shared memory are detected, may commit, or abort otherwise. Unlike lock based synchronization, transactions are composable and deadlock free.

Transactional memory implementations record the reads and writes that transactions perform and use this information to detect conflicts. A conflict occurs if two or more different transactions access the same memory location and at least one of these accesses is a write. Sometimes the conflict detecting mechanisms used by transactional memory systems are too conservative, and end up detecting false conflicts, or conflicts that might not violate the abstraction of a program. One classical example is two different transactions modifying different parts of a linked list [18,13,17,15]. Although their actions do not conflict, there is a read/write conflict as one transaction is modifying a memory location that was read by another transaction. This kind of read/write conflict detection scheme can have a huge impact in performance when using certain kinds of linked data structures or generally when accessing memory locations that are subject to high contention. On the other hand, using lock based synchronization, or even *lock-free* algorithms, expert programmers can achieve a high level of concurrency at the cost of code complexity. One alternative to combine these two worlds is *transactional boosting* [13]. Transactional boosting is a methodology used to transform highly concurrent linearizable objects into highly concurrent *transactional* objects. It treats base objects as black boxes: we can make a boosted version of a linearizable concurrent library with no knowledge on how it is implemented. Transactional boosting is not applicable to every problem,

F.M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 145–159, 2014.

basically, the methodology requires that method calls have *inverses*, and that only commutative method calls can occur concurrently.

STM Haskell [11] is a Haskell extension that provides the abstraction of *composable memory transactions*. The programmer defines *transactional actions* that are composable i.e., they can be combined to generate new transactions, and are first-class values. Haskell's type system forces threads to access shared variables only inside transactions. As transactions can not be executed outside a call to `atomically`, properties like *atomicity* (the effects of a transaction must be visible to all threads all at once) and *isolation* (during the execution of a transaction, it can not be affected by other transactions) are always maintained.

This text describes our experiments with transactional boosting in the context of the functional language Haskell. The contributions of this paper are as follows:

- We propose a new construct for STM Haskell that lets programmers implement boosted versions of existing fast concurrent Haskell libraries. We also describe how the new construct proposed here can interact with the high-level transactional primitives `retry` and `orElse` available in STM Haskell.
- We present the implementation of three classic transactional boosting examples using available concurrent Haskell libraries and the new proposed primitive. These examples demonstrate that our extension lets programmers transform existing fast implementations of linearizable structures into composable STM Haskell actions.
- Preliminary performance measurements are presented and indicate that, although transactional boosting can only be used in certain cases, when applied correctly, it is possible to achieve much faster STM actions than using the original STM Haskell. We believe that the primitive proposed here can be used by expert programmers do develop fast concurrent libraries for STM Haskell.

This paper is organized as follows: Section 2 reviews the main concepts of transactional memory and STM Haskell. Section 3 presents our extension to STM Haskell. Next, in Section 4, three examples of boosted versions of abstract data types are described and implemented using the new STM Haskell extension: a unique ID generator, a pipeline buffer, and a set data structure. Section 5 explains how the new primitive was implemented and shows the results of preliminary experiments. Finally, Section 6 discusses related work, and Section 7 concludes.

2 Transactional Memory and STM Haskell

2.1 Transactional Memory

Transactional memory was first described as a Hardware feature [12]. This paper focuses on *Software Transactional Memory* (STM), in which transactions are mainly implemented in software, with little hardware support, i.e., a compare and swap operation.

In an STM system, memory transactions can execute concurrently and, if finished without conflicts, a transaction may commit. Conflict detection may be *eager*, if a conflict is detected the first time a transaction accesses a value, or *lazy* when it occurs only at commit time. With eager conflict detection, to access a value, a transaction must acquire ownership of the value, hence preventing other transactions to access it, which is also called *pessimistic* concurrency control. With *optimistic* concurrency control, ownership acquisition and validation occurs only when committing. These design options can be combined for different kinds of accesses to data, e.g., eager conflict detection for write operations and lazy for reads. STM systems also differ in the granularity of conflict detection, being the most common word based and object based. In STM Haskell conflicts are detected at a TVar level, e.g, two different transactions writing to the same TVar (see next Section).

STM systems need a mechanism for version management. With *eager* version management, values are updated directly in memory and a transaction must maintain an *undo log* where it keeps the original values. If a transaction aborts, it uses the undo log to copy the old values back to memory. With *lazy* version management, all writes are buffered in a *redo log*, and reads must consult this log to see earlier writes. If a transaction commits, it copies these values to memory, and if it aborts the redo log can be discarded.

An STM implementation can be *lock* based, or *obstruction free*. An *obstruction free* STM does not use blocking mechanisms for synchronization and guarantees that a transaction will progress even if all other transactions are suspended. Lock based implementations, although offering weaker progress guarantees, are believed to be faster and easier to implement [8].

2.2 STM Haskell

STM Haskell extends Haskell with a set of primitives for writing memory transactions[11]. The main abstractions are *transactional variables* or `TVar`s, which are special variables that can only be accessed inside transactions. Figure 1 shows the main STM Haskell primitives. The `readTVar` function takes a TVar and returns a *transactional action* `STM a`. This action, when executed, will return a value of type `a`, i.e., the TVar's content. Similarly, `writeTVar` takes a value of type `a`, a TVar of the same type and returns a STM action that when executed writes into the TVar.

These transactional actions can be composed together to generate new actions using the basic monadic composition constructs (bind (`>>=`), then (`>>`) and `return`), or by using the syntactic sugar provided by the `do` notation.

The `retry` primitive is used to abort and re-execute a transaction once at least one of the memory positions it has read is modified. `orElse` is a composition operator, it takes two transactions as arguments, if the first one retries then the second one is executed. If both fail the whole transaction is executed again.

An STM action can only be executed with a call to `atomically`:

```
atomically (transferMoney tvar1 tvar2 100.00)
```

```
writeTVar :: TVar a -> a -> STM ()
readTVar :: TVar a -> STM a
retry :: STM ()
orElse :: STM a -> STM a -> STM a
atomically :: STM a -> IO a
```

Fig. 1. STM Haskell interface

It takes as an argument a transactional action (`STM a`) and executes it atomically with respect to other concurrently executing transactions.

3 Transactional Boosting for STM Haskell

Transactional boosting is a methodology used to transform existing highly concurrent linearizable objects into highly concurrent transactional objects. It treats existing objects as black boxes, and performs both conflict detection and logging at the granularity of entire method calls. It can not be applied to every object but to objects where commutative method calls can be identified, and which reasonably efficient inverses exist or can be composed from existing methods [13].

To write a transactional boosted version of an abstract type, the operations associated with this type must have inverses. Hence the STM system must provide ways of registering user defined handlers to be called when the transaction aborts or commits. If a transaction aborts, it must undo the changes it has done to the boosted object and if it commits it must make these changes visible to the rest of the system.

We propose a simple new STM Haskell primitive to build transactional versions of abstract data types:

```
boost :: IO (Maybe a) -> ((Maybe a)-> IO ()) -> IO () -> STM a
```

The `boost` primitive can be used to create a new transactional version of existing Haskell libraries. It wraps a function in a Haskell STM action, providing ways to call this function inside a transaction and also for undoing its effects in the case of an abort. It takes as arguments:

- an *action* (of type `IO (Maybe a)`) that is used by the underlying transactional system to invoke the original function. When the original function is called it might return a result of type `a`, or maybe for some reason the original function could not be called, e.g., an internal lock could not be acquired, in which case the action should return `Nothing`. Hence the return type is `Maybe a`
- an *undo* action (of type `Maybe a -> IO ()`) used to undo the function call in case of an abort. If we want to abort a `STM` action, we must know what was the outcome of executing it, e.g., if we want to undo deleting value x from a set, we must insert x again into the set. As the outcome of executing a transactional action is only known at execution time, the `undo` action takes as argument the value returned by the first argument of `boost`.

– a *commit* action (of type `IO ()`) that is used to commit the action done by the boosted version of the original function, i.e., make it visible to the rest of the system

`boost` returns a new `STM` action that is used inside transactions to access the wrapped function.

4 Examples

This section describes the implementation of three classic transactional boosting examples using existing concurrent Haskell libraries plus the new primitive proposed for STM Haskell. The examples are a *Unique ID Generator* (Section 4.1), a *Pipeline Buffer* (Section 4.2), and a *Set* (Section 4.3).

4.1 Unique ID Generator

We start with a simple example, a unique ID generator. Generating unique IDs in STM systems is problematic. The `generateID` function would typically be implemented using a shared counter that is incremented at each call. As different transactions are trying to increment and read a shared location (the counter), transactional memory implementations detect read/write conflicts and abort at least one of the transactions. The problem is that those may not be real conflicts: as long as different calls to `generateID` return different numbers, we do not care, for example, if the values generated follow the exact order in which the counter was incremented.

A simple and fast thread safe unique ID generator could be implemented using a Fetch-and-Add or Compare-and-Swap (CAS) operation, available in most standard multi-core processors. Haskell, provides an abstraction called `IORef` that represents a mutable memory locations [16]. An `IORef a` is a mutable memory location that may contains a value of type `a`. The library [3] lets programmers perform machine-level compare and swap operation on `IORef`s. Thus, a unique ID generator can be implemented as follows:

```
type IDGer = IORef Int

newID :: IO IDGer
newID = newIORef 0

generateID ::  IDGer -> IO Int
generateID idger = do
    v <- readIORef idger
    ok <- atomCAS idger v (v+1)
    if ok then return (v+1) else generatetID idger
```

Under transactional boosting, the ID generator must follow the specification in Figure 2.

Function Call	Inverse
`generateID`	noop

Commutativity

`x <-generateID` $\Leftrightarrow$ `y <-generateID` $x \neq y$
`x <-generateID` $\not\Leftrightarrow$ `y <-generateID` $x = y$

Fig. 2. Unique ID Generator specification

When a transaction that called `generateID` aborts, ideally the ID returned by the call should be returned to a pool of unused IDs. On the other hand, since `generateID` always returns an unique value, the IDs generated are disposable.

Using transactional boosting, the `generateID` function could be implemented as follows:

```
generateIDTB ::  IDGer -> STM Int
generateIDTB idger =  boost ac undo commit
     where
      ac = do
         id <- UniqueIDCAS.generateID idger
         return (Just id)
      undo _ = return ()
      commit = return ()
```

Now `generateIDTB` is an `STM` action that, when executed calls the `ac` action which simply uses the CAS version of the generator to increment the `IORef` counter. If the transaction aborts, or the transaction commits, nothing has to be done, hence the `undo` and `commit` actions are empty.

4.2 Pipeline Buffer

Pipeline is an abstraction where there is a chain of data processing threads that communicate by bounded queues, or buffers. Each thread is in charge of a stage in the pipeline and consumes data from a buffer, processes it, and writes the new data to a different buffer.

A buffer must provide two functions: `offer` used to add a value to the buffer and `take`, that consumes a data item. To implement a buffer using transactional

Function Call	Inverse
`offer buf x`	tryPopL buf
`x <-take buf`	pushR buf x

Commutativity

`offer buf x` $\Leftrightarrow$ `y <-take buf`, buffer non-empty
`offer buf x` $\not\Leftrightarrow$ `y <-take buf`, otherwise

Fig. 3. Pipeline Buffer specification

boosting, we need a double-ended queue as it provides inverses for `take` and `offer`. Here we use a thread safe double-ended queue implemented by Ryan Newton [4], and follow the specification in Figure 3:

```
data TBBuffer a = Q (SimpleDeque a) (IORef Int)

newTBBuffer :: TBBuffer a
newTBBuffer = do
      q<-newQ
      ioref <- newIORef 0
      return (Q q ioref)

offer :: TBBuffer a -> a -> STM ()
offer (Q c ioref) v = boost ac undo commit
      where
        ac = do
           pushL c v
           return (Just ())
        undo _ = do
             mv <- tryPopL c
             case mv of
                 Just v -> return ()
        commit = do
             v <- readIORef ioref
             ok<- atomCAS ioref v (v+1)
             if ok then return () else commit
```

A `TBBuffer` is represented by a double-ended queue and an `IORef` that contains the size of the buffer. The `offer` function must use `pushL` to add a new value to the queue. If a transaction aborts, the data that was inserted in the queue must be eliminated using `tryPopL`. If the transaction commits, the only thing to be done is to increment the buffer size thus making the new item visible to the transaction that is consuming values. The `take` function uses `tryPopR` to consume data from a buffer:

```
take :: TBBuffer a -> STM a
take (Q c ioref) = boost ac undo commit
   where
     ac = do
        size<-readIORef ioref
        if size ==0 then return Nothing
                    else do
                       decIORef ioref
                       tryPopR c
     undo v = case v of
                Nothing -> return ()
                Just x -> do
                       incIORef ioref
                       pushR c x
     commit = return ()
```

If there are not enough items then the transaction aborts by returning `Nothing`. Otherwise it decrements the buffer counter using CAS (with the `decIORef` function) and consumes an item (`tryPopR`). The `undo` action must increment the counter using CAS (`incIORef`) and return the value taken back to the buffer.

4.3 Set

A set is a collection of distinct objects. An implementation of a set usually provides three functions, `add`, `remove` and `contains`.

Function Call	Inverse
`add set x / False`	noop
`add set x / True`	`remove set x / True`
`remove set x / False`	noop
`remove set x / True`	`add set x / True`
`contains set x / _`	noop

Commutativity

`add set x / _` $\Leftrightarrow$ `add set y / _`, $x \neq y$

`remove set x / _` $\Leftrightarrow$ `remove set y / _`, $x \neq y$

`add set x / _` $\Leftrightarrow$ `remove set y / _`, $x \neq y$

`add set x / False` $\Leftrightarrow$ `remove set x / False` $\Leftrightarrow$ `contains set x / _`

Fig. 4. Set specification

As there is no linearizable implementation of a set data structure available for Haskell, the boosted set described here uses a thread safe linked list, described in [18] (see Figure 5). When implementing a boosted version of a set, we must guarantee that if one transactions is working on a key, no other transaction can be using the same key (see Figure 4). We can achieve this by using key-based locking [15]. Key based locking can be implemented using a hash table to associate a lock with each key. The problem is that currently there are no thread safe Hash tables available for Haskell. To implement key locking we use an STM Hash table from the Haskell STM benchmark suite [1]:

```
data KLock = KLock (THash Int Lock)

data Lock = Lock (IORef Integer) (IORef Integer)

newKLock :: IO KLock
newKLock = do
      hasht <- atomically (new hashInt)
      return (KLock hasht)
```

A `KLock` is a hash table that maps `Int`s (keys) to locks. Locks are represented by two `IORef`s. The first is a versioned lock [9]: if it contains `0` the locks is free, if it contains a transaction ID the lock is locked. The second `IORef` counts how

many times the lock holder locked the same key. The `newKLock` function creates a new `KLock` by simply creating an empty Hash table. A key can be locked with the `lock` function:

```
lock :: KLock -> Int -> IO Bool
lock (KLock ht) key = do
   mior <- atomically (Data.THash.lookup ht key)
   myId <- getTransID
   case mior of
       Just (Lock ior counter) -> do
                 v <- readIORef ior
                 if (v == 0)
                     then do locked<- atomCAS ior 0 myId
                             case locked of
                                   True -> do
                                      plusOne counter
                                      return True
                                   False -> return False
(...)
```

It takes a `Klock` and a `key` as arguments. If there is already a lock associated with the key, and the lock is free (i.e., contains `0`), it tries to acquire the lock using `atomCAS`. If successful, it increments the counter. If the current transaction already holds the lock, it just needs to increment the counter one more time. If there is no lock associated with the key, a new one must be created and inserted into the hash table:

```
       Nothing -> do
                 ior <- newIORef myId
                 counter <- newIORef 1
                 ok <- atomically (insert ht key (Lock ior counter))
                 if ok then return True else (lock alock key)
```

The `unlock` function

```
unlock :: KLock -> Int -> IO Bool
```

finds the lock associated with the key, and if the current transaction holds the lock it simply decrements the lock's counter. If the counter gets to zero, then the lock is freed using CAS.

```
newList :: IO (ListHandle a)
addToTail :: Eq a => ListHandle a -> a -> IO ()
find :: Eq a => ListHandle a -> a -> IO Bool
delete :: Eq a => ListHandle a -> a -> IO Bool
```

Fig. 5. Interface for the concurrent linked list described in [18]

Now, a boosted version of a `Set` data structure can be represented by a `KLock` and a linked list:

```
data IntSet = Set KLock (ListHandle Int)
```

To `add` a `key` to a set, we must acquire the lock associated with it and then insert the key in the linked list. As the linked list may contain duplicates, we must also make sure that the `key` is not already in the list:

```
add:: IntSet -> Int -> STM Bool
add (Set klock list) key =  boost ac undo commit
   where
     ac =  do
        ok<-lock klock key
        case ok of
           True -> do found <- CASList.find list key
                      if found then return (Just False)
                               else do
                                   CASList.addToTail list key
                                   return (Just True)
           False -> return Nothing
```

If a key was inserted and the transaction aborts, the same key must be deleted from the list and the lock freed. If the transaction commits, the only thing to do is to free the lock:

```
     undo v = do
        case v of
           Just True -> do
                CASList.delete list key
                unlock klock key
                return()
           Just False ->  do
                unlock klock key
                return()
           Nothing -> return ()

     commit = do
        unlock klock key
        return ()
```

To `remove` a key, we must acquire the right lock and then delete the key from the linked list:

```
remove :: IntSet -> Int -> STM Bool
remove (Set klock list) key = boost ac undo commit
    where
     ac =  do
        ok<-lock klock key
        case ok of
           True -> do v<-CASList.delete list key
                      return (Just v)
           False -> return Nothing
     undo ok = do
        case ok of
           Just True -> do
```

```
                CASList.addToTail list key
                unlock klock key
                return ()
          Just False-> do unlock klock key
                          return ()
          Nothing -> return ()
    commit = do unlock alock key
                return ()
```

To `undo` a successful `remove`, the `key` must be inserted back again in the list and the lock freed. As before, to `commit` the operation we just need to liberate the lock.

The `contains` operation

```
contains :: IntSet -> Int -> STM Bool
```

is simpler and the code is omitted. As `contains` does not changes the internal list, if the transaction aborts or commits, nothing has to be done except liberating the lock associated with the searched key.

5 Implementation and Preliminary Experiments

5.1 Prototype Implementation

To test the examples presented in this paper, we extended TL2 STM Haskell [7,2], a high-level implementation of STM Haskell that uses the TL2 [6] algorithm for transactions. The TL2 implementation uses lazy conflict detection with optimistic concurrency control as happens in the original C implementation of STM Haskell that comes with the GHC compiler[10].

As the TL2 library is implemented completely in Haskell, it is easier to extend and modify. It also provides reasonable performance for a prototype: programs run 1 to 16 times slower than the C implementation, with factors of 2 and 3 being the most common. Experiments using the Haskell STM Benchmark suite also demonstrate that the library provides scalability similar to the original C run-time system [7].

In TL2 STM Haskell, as in other implementations of STM Haskell [14,5], the STM data type is represented as a state passing monad. Thus, an `STM a` action in the monad is in fact a function that takes the state of the current transaction (e.g. its read and write logs) as an argument, executes a computation in the transaction (e.g. reads a `TVar`), and returns a new transaction state and a result of type `a`:

```
data STM a = STM (TState -> IO (TResult a))
```

The `TResult` type describes the possible outcomes of executing a transaction:

```
data TResult a = Valid TState a | Retry TState | Invalid TState
```

Our implementation of the `boost` extends the type that represents the transaction state (`TState`) with two `IO ()` actions:

```
data TState = Tr {
    (...)
   tbUndo :: IO (),
   tbCommit :: IO ()
}
```

The `tbUndo` and `tbCommit` actions are constructed during the execution of a transaction, and the first is executed only if a transaction aborts, i.e., finishes with an `Invalid` state, and the later is executed only if a transaction commits. Hence, the implementation of `boost` becomes simply:

```
boost :: IO (Maybe a) -> (Maybe a-> IO ()) -> IO () -> STM a
boost mac undo commit = STM $ \tState -> do
  r <- mac
  case r of
    Just v -> return (Valid tState{tbUndo=undo (Just v)>>(tbUndo tState),
                                   tbCommit = commit>>(tbCommit tState)} v)
    Nothing -> return (Invalid tState{tbUndo=undo Nothing>>(tbUndo tState)})
```

The *then* monadic combinator (`>>`) is used to add the new actions to the current `tbUndo` and `tbCommit` IO actions.

The design and implementation of the `retry` construct is tied closely to the concept of `TVar`: when a transaction retries, it will not execute again until at least one of the tvars it read is updated. We would like to extend the concept further, for example, in the `Set` example, if we can not acquire the lock for a key, we could retry or abort the transaction and only start it again once the lock is freed. It is difficult to predict all possible cases in which we want a transaction to be restarted and we also do not want to include in the design primitives that are too low level. Hence we decided for a simpler approach: when a transaction that executed TB actions calls `retry`, the transaction is stopped, `tbUndo` is executed, and the transaction waits on the `TVars` it has read.

We also extend the behavior of the `orElse` construct to support transactional boosting. A call to `orElse t1 t2` will first save the `tbUndo` and `tbCommit` actions of the current transaction's state and execute `t1` with new and empty `tbUndo` and `tbCommit`. If `t1` retries, its newly created `tbUndo` is called and the transaction continues by executing `t2` with the saved actions. If `t1` finishes without retrying, then both new and saved TB actions are combined and `orElse` returns the result of executing `t1`.

5.2 Experiments

The experiments were executed on an Intel Core i7 processor at 2.1 GHz and 8 GiB of RAM and the times presented are the mean of 10 executions. The operating system was Ubuntu 12.04, with Haskell GHC 7.4. The Core i7 processor has 4 real cores plus 4 more with hyper threading.

Figure 6 compares three implementations of the unique ID generator: `ID STM` that uses the C implementation of STM Haskell that comes with GHC, `ID CAS` that uses CAS to increment the ID counter, and `ID TB` that is the boosted implementation described in Section 4.1. The experiment executes 10 million calls to `generateID` in total, dividing these operations by the threads available.

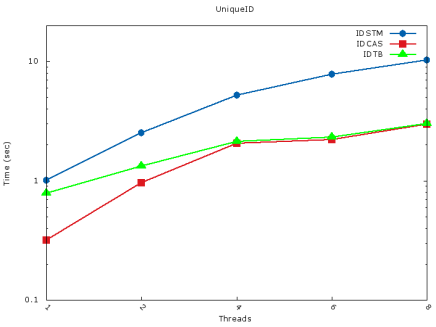


Fig. 6. UniqueID execution times

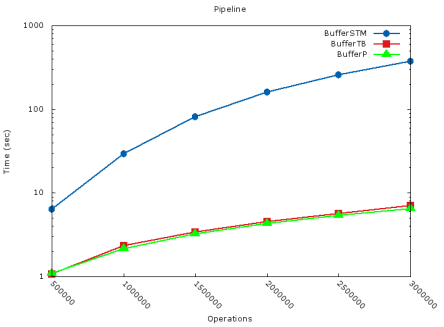


Fig. 7. Pipeline Buffer execution times

The second experiment (Figure 7) creates two threads, a producer and a consumer, that communicate through a pipeline buffer each executing the number of operations described on the X axis. Three different implementations of the pipeline buffer are compared: `Buffer STM` that uses the `TChan` STM library that comes with GHC, `BufferP` is the thread safe deque implemented by Ryan Newton, and `BufferTB` is the boosted version described in Section 4.2.

To benchmark the set data structure, we randomly generated test data consisting of an initial list of 2000 elements and 8 lists of 2000 operations. Each list of operations is given to a different thread and we vary the number of cores used from 1 to 8, as can be seen in Figure 8. Note that the Y axis is a logarithmic scale. Two implementations of Set are compared, one that uses an ordered linked list of TVars and is compiled using the original C STM Haskell (`GHC-STM`), and the boosted version described in Section 4.3 (`TB`).

As can be seen by these preliminary experiments, even though our prototype implementation uses the slower TL2 STM Haskell implementation, all TB examples are still much faster than the original STM Haskell implemented in C. This happens because the overhead introduced by transactional layer added to the fast parallel implementations is not enough to harm the performance.

6 Related Work

Other works have extended the original STM Haskell design with *escape* primitives that allow the programmer to change the state of the current transaction hence avoiding false conflicts. The `unreadTVar` [17] construct is an extension to the original STM Haskell interface that improves execution time and memory

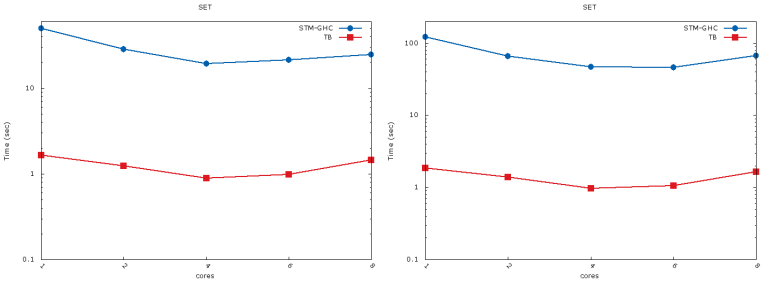


Fig. 8. 8 threads each executing 2000 operations in a Set (On the left: 40% `add`s and `remove`s + 60% `contain`s, On the right: 65% `add`s and `remove`s + 25% `contain`s)

usage when traversing transactional linked structures. It is used to eliminate from the read set values that are far behind from the current position. For the same purpose in [18] the authors propose `readTVarIO` that reads a TVar without adding a log entry for this read into the current transaction's read set. If the lock of the TVar is being held by a transaction trying to commit, `readTVarIO` blocks until the lock is freed. *Twilight* STM in Haskell [5] extends STM Haskell with safe embedding of I/O operations and a repair facility to resolve conflicts. Twilight splits the code of a transaction into two phases, a functional atomic phase (that behaves just like the original STM Haskell), and an imperative phase where Twilight code can be used. A prototype implementation of Twilight Haskell in Haskell is provided but no performance figures are given.

All these new primitives are used to implement new data types in a way that false conflicts are avoided. The method presented here is used as a way of accessing existing fast highly concurrent structures inside transactions.

7 Concluding Remarks

We have described an STM Haskell extension to write transactional boosted versions of abstract data types. We extended STM Haskell with a single new primitive and have described three examples of its use. Preliminary experiments show that the new primitive can help programmers to write faster transactional libraries if used in the right context. Transactional boosting is low level concurrent programming and can lead to all the problems associated with concurrency, such as deadlocks [13]. We believe that the primitive presented here can be used by expert programmers to write fast concurrent libraries for Haskell.

We can also use `boost` to implemented transactional versions of sequential structures that are not available in STM libraries, however there will be a performance impact depending on the approach used to protect these structures (e.g., a single lock) plus the overhead of the transactional boosting system, as can be seen in the experiments in Section 5.

References

1. The Haskell STM Benchmark. WWW page (October 2010), http://www.bscmsrc.eu/software/haskell-stm-benchmark
2. TL2 STM Haskell (February 2011), http://sites.google.com/site/tl2stmhaskell/STM.hs
3. Data.CAS (October 2013), http://hackage.haskell.org/package/IORefCAS-0.1.0.1/docs/src/Data-CAS.html
4. Thread Safe Deque (October 2013), https://github.com/rrnewton/haskell-lockfree
5. Bieniusa, A., et al.: Twilight in Haskell: Software Transactional Memory with Safe I/O and Typed Conflict Management. In: IFL 2010 (September 2010)
6. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
7. Du Bois, A.R.: An implementation of composable memory transactions in Haskell. In: Apel, S., Jackson, E. (eds.) SC 2011. LNCS, vol. 6708, pp. 34–50. Springer, Heidelberg (2011)
8. Ennals, R.: Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report (January 2006)
9. Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory, 2nd edn. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers (2010)
10. Harris, T., Marlow, S., Peyton Jones, S.: Haskell on a shared-memory multiprocessor. In: Haskell Workshop 2005, pp. 49–61. ACM Press (September 2005)
11. Harris, T., Marlow, S., Peyton Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP 2005. ACM Press (2005)
12. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA, pp. 289–300 (May 1993)
13. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: PPoPP, pp. 207–216. ACM (2008)
14. Huch, F., Kupke, F.: A high-level implementation of composable memory transactions in concurrent Haskell. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 124–141. Springer, Heidelberg (2006)
15. Ni, Y., et al.: Open nesting in software transactional memory. In: PPoPP, vol. 1, pp. 68–78. ACM (2007)
16. Peyton Jones, S.: Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: Engineering Theories of Software Construction, pp. 47–96. IOS Press (2001)
17. Sonmez, N., et al.: Unreadtvar: Extending Haskell software transactional memory for performance. In: Trends in Functional Programming. Intellect Books (2008)
18. Sulzmann, M., Lam, E.S., Marlow, S.: Comparing the performance of concurrent linked-list implementations in Haskell. SIGPLAN Not. 44(5), 11–20 (2009)

Author Index