

# The Design of CoreC

Greta Yorsh      Nurit Dor

Mooly Sagiv,

School of Computer Science, Tel-Aviv University, Israel

April 12, 2003

## Abstract

This manuscript is intended as a reference manual for the C code *Simplifier*. The output of *Simplifier* is a C program with limited form of expressions, but with extra temporary variables, called *CoreC*.

## 1 Introduction

In this chapter we address a practical aspect of program analysis — dealing with real applications coded in C. Towards this end, we develop *Simplifier*, a tool that transforms a C program into an equivalent C program that uses only a small subset of the C language, called *CoreC*. This tool enables faster development of source-code analyzers; it was used by CSSV [DRS03] to check real C programs for string errors. The novelty of our work is the design of *CoreC* and the meaning-preserving translation rules from C to *CoreC*.

The main features of *CoreC* are:

- The *CoreC* syntax is compact and simple: (i) the number of different constructs that may appear in a *CoreC* program is relatively small; (ii) each construct has only one operation and simple arguments. This syntax makes it easy to specify the semantics of *CoreC* programs, because each statement performs a single operation.
- Due to this, *CoreC* makes it easy to define and implement static analyses and source-to-source transformations of C programs. All that is needed to handle the full C language is to specify the semantics for *CoreC* constructs. The reason is that the translation rules preserve the semantics of the original program.
- *CoreC* is an executable intermediate representation, which enables easy debugging of *CoreC* itself.
- As opposed to many other intermediate representations, *CoreC* is portable across different machines and compilers.

The rest of the chapter is organized as follows: Section 2 defines a subset of C called *CoreC*. Source-to-source translation rules for converting C into *CoreC* are described in Section 3. Related work is described in Section 5. Finally, the limitations of the current translation are listed in Section 4.

## 2 The *CoreC* Language Subset

The *CoreC* language is a subset of C with the following restrictions:

- The only control-flow constructs are `goto` and an if-statement with `goto`s in the then- and else-clauses.
- Expressions cannot have side effects. In particular, assignments are statements. Expressions such as `+=`, `++`, and `comma` expressions are not allowed.
- There are two types of expressions: (i) Boolean comparison expressions used in if-conditions, and (ii) “Value” expressions (arithmetic, dereferences, and address expressions), which are used as L-values and R-values in assignments. Both types of expressions are rather limited. They cannot include more than one operation. The field operator (`a . b`) is not considered as an operation, and therefore can appear more than once in an expression. Dereference (`*`) is considered as an operation. The only exception to this rule is the address operator, which can be applied to a dereference: `& (*a) . b`. Moreover, at most one side of an assignment can include an operation. Value expressions cannot include comparison operators. Finally, logical connectives `&&`, `||`, and `!` are not allowed at all.
- Variables can only be initialized through regular assignments; initializations in declarations are not allowed. As a consequence, variables defined with `const` type qualifiers are not allowed, because they cannot be initialized, except in a declaration. However, `const` qualifiers can be safely removed, because we assume that the original program passes compilation, which guarantees that the program contains no assignments to `const` variables.

The syntax of the *CoreC* language is given in Table 1. The syntax is in extended BNF. IDENTIFIER and CONSTANT represent valid identifier and constant symbols, accepted by standard C compilers. The syntax follows the definitions given in [Deg95] and [KR78]. Section 5 compares *CoreC* syntax to other representations.

```

<CoreC> ::= (<functionDefinition> | <declaration>)*
<declaration> ::= <declarationSpecifiers> [<declarator> ]
<functionDefinition> ::= [<declarationSpecifiers> ]<declarator>
                        <declaration> * <compoundStatement>
<declarationSpecifiers> ::= (<storageClassSpecifier> | <typeSpecifier>)*
<storageClassSpecifier> ::= TYPEDEF | EXTERN | STATIC | AUTO | REGISTER
<typeSpecifier> ::= VOID | CHAR | SHORT | INT | LONG | FLOAT | DOUBLE | SIGNED |
                  UNSIGNED | <structOrUnionSpecifier> | <enumSpecifier> | typeName
<structOrUnionSpecifier> ::= (STRUCT | UNION) [IDENTIFIER]{<structDeclaration>+ }
<structDeclaration> ::= <specifierList> <structDeclarationList>
<specifierList> ::= (<typeSpecifier>)+
<structDeclarationList> ::= <structDeclarator> (, <structDeclarator>)*
<structDeclarator> ::= <declarator> |
                    <declarator> : <constantExpression>
<enumSpecifier> ::= ENUM [IDENTIFIER][{<enumeratorList> }]
<enumeratorList> ::= <enumerator> (, <enumerator>)*
<enumerator> ::= IDENTIFIER [= <constantExpression> ]
<declarator> ::= [<pointer> ]<directDeclarator>
<pointer> ::= (*)+
<directDeclarator> ::= IDENTIFIER |
                    <directDeclarator> ( [<declaration> ]* , [...]) // function
                    | <directDeclarator> [CONSTANT] // array

```

Table 1: The *CoreC* syntax.

<pre> &lt;compoundStatement&gt; ::= {&lt;declaration&gt;* &lt;statement&gt;* } &lt;statement&gt; ::= &lt;compoundStatement&gt;                   IDENTIFIER : &lt;statement&gt;                   &lt;Assignment&gt;;                   IF ( &lt;booleanExpression&gt; ) goto IDENTIFIER; else goto IDENTIFIER;                   GOTO IDENTIFIER ;                   RETURN [&lt;primaryExpression&gt; ];                   &lt;Call&gt;;                   ; // empty statement &lt;Assignment&gt; ::= &lt;name&gt; = &lt;simpleExpression&gt;                   &lt;deref&gt; = &lt;primaryExpression&gt; &lt;deref&gt; ::= (* &lt;name&gt;).(FIELD_NAME)* &lt;simpleExpression&gt; ::= &lt;primaryExpression&gt;                       (typeName) &lt;primaryExpression&gt; // cast                       &lt;primaryExpression&gt; &lt;binop&gt; &lt;primaryExpression&gt;                       &lt;unop&gt; &lt;primaryExpression&gt;                       &lt;deref&gt;                       &lt;address&gt;                       &lt;Call&gt; &lt;address&gt; ::= &amp;&lt;name&gt;   &amp;&lt;deref&gt; &lt;Call&gt; ::= IDENTIFIER ( [&lt;argList&gt; ] ) &lt;argList&gt; ::= primaryExpression [, primaryExpression ]* &lt;booleanExpression&gt; ::= &lt;primaryExpression&gt; &lt;relop&gt; &lt;primaryExpression&gt; &lt;name&gt; ::= IDENTIFIER ( .FIELD_NAME )* &lt;primaryExpression&gt; ::= &lt;name&gt;   CONSTANT &lt;relop&gt; ::= ==   !=   &lt;=   &gt;=   &lt;   &gt; &lt;binop&gt; ::= +   -   *   /   %   ^   &lt;&lt;   &gt;&gt;       &amp; &lt;unop&gt; ::= +   -   ~ </pre>
--

Table 1: The *CoreC* syntax.

### 3 From C into *CoreC*

A C program is converted into a *CoreC* program by translating complex expressions into expressions with at most one operation. In certain cases, new temporary variables are introduced to store intermediate values. The resulting *CoreC* program is equivalent to the original C program in the sense that if the C program is defined according to ANSI-C, both C and *CoreC* programs provide the same output.

**Example 3.1** Fig. 1 shows an example program *SkipLine* and its translation to *CoreC*. Note that double pointer indirections are translated into two

sequential pointer indirections and that the `for` loop is translated into an `if-goto` loop.

This section provides a high-level description of the algorithm for converting C to *CoreC*. It is based on the algorithm for translating abstract syntax into abstract machine code, used in compilers, as described in Chapter 7 of [App98]. To provide a front-end-independent description of the translation process, we assume that an abstract syntax tree (AST) of the input C program is given in a form that allows top-down traversal and property retrieval. At the end of the traversal of the input AST, an equivalent AST is generated, containing only legal *CoreC* constructs. The translation process recursively traverses each sub-expression of an AST node, generates equivalent *CoreC* constructs for each subexpression, and combines them.

### 3.1 Internal structure

To define the translation inductively, the algorithm uses an internal structure named *CoreC* Unit (CCU). CCU accumulates declarations and statements created during traversal. CCU also contains sub-expressions to be combined in a higher level. The CCU structure has a similar functionality to the `Tr_exp` structure in [App98]. The differences are due to the complexity of C language, compared to Tiger. As in `Tr_exp`, there are several kinds of CCU structures (see Table 2):

**CCUExp** - *CoreC* expression.

**CCUBool** - Boolean expression, contains only one relational or equality operator, applied to primary\_expressions.

**CCUArgList** - list of *CoreC* expressions, to be used as arguments to a *CoreC* function call.

**CCUCond** - if a Boolean expression can be used either as a value or in a conditional jump (i.e. the translation depends on the context), the decision is delayed and conditional unit **CCUCond** is generated. The Boolean expression is used as a condition in if-statement. This if-statement is added to the statement list in the basic part of CCU. Two new labels are generated for true and false clauses of this if-statement. Each label is added to the suitable list of labels in **CCUCond**. Now, it is the responsibility of the caller to "patch" the labels at the right places (either where if-branches start or where the value of the Boolean expression is assigned to a new temporary, for future use in an expression).

**CCUStmt** - holds two list of labels, referenced by the statements of the unit, but not yet located. When the translation of current unit is finished, the labels will be added before and after the statements of this unit. This "patch" is handy for translating "break" and "continue" statements used within loops and switch.

<pre> # define SIZE 5 void SkipLine(int NbLine,               char** PtrEndText) {     int indice;     for (indice=0; indice&lt;NbLine;          indice++)     {         **PtrEndText = '\n';         (*PtrEndText)++;     }     **PtrEndText = '\0'; }  void main() {     char buf[SIZE]="a\nb";     char *r, *s;      r = buf;     SkipLine(1,&amp;r);     fgets(r,SIZE-1,stdin);     s = r + strlen(r);     SkipLine(1,&amp;s); } </pre> <p style="text-align: center;">6</p> <p style="text-align: center;">(a)</p>	<pre> void SkipLine(int NbLine,               char** PtrEndText) {     int indice; char *tmp4;     indice=0; begin_loop:     if (indice&gt;=NbLine)         goto end_loop;     else goto loop_body; loop_body:{     char *tmp1; char *tmp2; char *tmp3;     tmp1 = *PtrEndText     *tmp1 = '\n';     tmp2 = *PtrEndText;     tmp3 = tmp2 + 1;     *PtrEndText = tmp3;     }     indice = indice + 1;     goto begin_loop; end_loop:     tmp4 = *PtrEndText     *tmp4 = '\0'; }  void main() {     char buf[5];     char *r; char *s;     char **tmp4; char **tmp7;     int tmp5; int tmp6;      buf[0]='a';     buf[1]='\n';     buf[2]='b';     buf[3]='\0';     buf[4]='\0';      r = buf;     tmp4 = &amp;r;     SkipLine(1,tmp4);     tmp5 = SIZE-1;     fgets(r,tmp5,stdin);     tmp6 = strlen(r);     s = r + tmp6;     tmp7 = &amp;s;     SkipLine(1,tmp7); } </pre> <p style="text-align: center;">(b)</p>
---	--

Figure 1: Example: (a) SkipLine, a string-manipulation function from EADS Airbus with a toy main program; (b) the result of translation SkipLine program

### 3.2 Value/Address computations

Sometimes we will have CCU of one kind and we will need to convert it to an equivalent unit of another kind. For example,  $f = (a > b) || (c < d)$  requires the conversion of CCUCond into CCUExp, so that the "true" or "false" results of computing  $||$  can be stored into "f". In general, the translation process may convert a value computation into an address computation and visa versa. To convert a "conditional" into a value, we invent a new temporary variable and assign "1" to it. This assignment is added to the statement list of current unit, preceded by labels from the "true" list of CCUCond. We handle the "false" branch similarly. The result is just the temporary.

### 3.3 L-value and R-value

Translation of an expression depends on whether it is an L-value or an R-value. When an expression is an R-value it denotes a value fetched from a location (i.e., the contents). When an expression is an L-value it denotes a location (i.e., an address) in which a value can be stored (for example, when the expression appears on the left-hand side of an assignment). Replacing an L-value expression by a temporary variable may change the meaning of the program (for example, see the use of  $a.f$  in  $a.f = b + c, b = \&(a.f)$ ). Therefore, we need special translation rules for expressions that serve as L-values.

### 3.4 Translation rules

Main modules of the algorithm are listed in Table 3. Each module takes an arbitrary C construct and produces a *CoreC* unit, which is semantically equivalent to the input construct. This *CoreC* unit contains *CoreC* declarations and statements generated during the simplification process of the input construct, including new variables and their initializations.

Informal pseudo-code description of the translation algorithm can be found in [Yor02]. In that document, some informal naming conventions are used to save space: *exp* denotes an expression, *stmt* denotes a statement, "*e.f*" denotes symbolic structure of expression, + denotes concatenation of elements of the same type into a list. Also note the function IsCore(*exp*, *core\_type*), that checks whether the expression *exp* is a legal *CoreC* expression of type *core\_type*, as defined in Table 1.

<i>CCUStmt simplify_stmt(stmt);</i> <i>Input:</i> C statement.
<i>CCUExp simplify_lvalue(exp);</i> <i>Input:</i> Arbitrary C expression denoting an L-value Generates an error when <i>exp</i> does not denote an L-value.

Table 3: Main modules of the translation algorithm.

<p><i>CCU simplify_exp(exp);</i>  <i>Input:</i> Arbitrary C expression denoting an R-value</p>
<p><i>CCU simplify_bool_exp(exp);</i>  <i>Input:</i> C expression representing a Boolean value  Generates either CCUCond or CCUExp.</p>
<p><i>CCUSmt simplify_decl(d);</i>  <i>Input:</i> an arbitrary C declaration with/without initialization.  <i>Output:</i> <i>CoreC</i> declaration equivalent to the input declaration, with the following exceptions: (a) removing "const" qualifiers; (b) initialization is converted into an assignment, which is added to the statement list.</p>
<p><i>CCUExp simplify_post(exp, bin_op);</i>  <i>CCUExp simplify_pre(exp bin_op);</i>  <i>Input:</i> C-expression denoting post-inc (x++) or post-dec(x-) operation and a suitable binary operator (+ or -).  Generates an error when exp does not denote an l-value.</p>
<p><i>CCUExp simplify_unary(exp, un_op, SymbolTable *scope);</i>  <i>Input:</i> C-expression denoting the operand of a unary operator un_op from the following list: ~e -e +e</p>
<p><i>CCUExp simplify_binary(e1, e2, bin_op);</i>  <i>Input:</i> C-expressions e1, e2 denoting operands of a binary operation: + - */%!&amp; &lt;&lt;&gt;&gt; according to bin_op param.</p>
<p><i>CCUExp simplify_op_assign(e1, e2, bin_op);</i>  <i>Input:</i> C expressions e1, e2 of and assignment "e1 op e2", where op is one of the following: +=, -=, *=, /=, bin_op denotes binary operator suitable for the assignment. Output: Expression equivalent to the value returned by the assignment after the evaluation, equivalent <i>CoreC</i> assignment, and additional declarations and statements created in the simplifier process.</p>

Table 3: Main modules of the translation algorithm.

## 4 Limitations of the Current Translation

We implemented the translation algorithm described above, using Microsoft AST Toolkit [TSBTG] as a frontend and a backend. This tool builds an abstract syntax tree of a program and allows traversal and modification of the tree. It also produces an output of the (modified) abstract syntax tree in the form of a C program.

**Explicit casting.** Signed/unsigned warning in the simplified code that is not produced by the original code. For example, consider the program fragment on Fig. 2(a) and its translation on Fig. 2(b). The last line of the translation produces a signed/unsigned mismatch.

**Initialization lists** - limited treatment for (i) casting of list elements when types do not match; (ii) nested initialization lists with missing elements.

```

CCU { CoreDeclarations d; CoreStatements s; }
CCUStmt extends CCU { LabelList start_labels, end_labels; }
CCUExp extends CCU { CoreExpression exp; }
CCUBool extends CCU { CoreBoolean exp; }
CCUArgList extends CCU { CorePrimaryList list; }

```

Table 2: CCU (*CoreC* Unit) is a structure used for translation

<pre> unsigned int a, b; if (n &lt; a/b) ... </pre> <p style="text-align: center;">(a)</p>	<pre> int n; unsigned int a, b, tmp; tmp = a/b; if (n &lt; tmp) ... </pre> <p style="text-align: center;">(b)</p>
--	---

Figure 2: Example: (a) a program fragment and (b) its translation to *CoreC*.

Simplifier may fail in some cases even if the initialization is correct according to ANSI C. It is recommended to correct the original code by adding explicit casting, missing elements, and nested lists for initialization of structure fields.

**Erroneous output** might be produced for the following features:

- variable-length parameter lists
- set jmp/long jmp
- signals
- system calls such as `system`, `exec`, `abort`, etc.

**sizeof** expression is replaced by a constant, which may be machine dependent.

**Optimizations:** The current translation rules do not produce minimal size *CoreC* code. It is possible to minimize the number of new variables, labels, and statements introduced during translation, by reusing temporaries, for example.

## 5 Related Work

There are numerous tools that convert C to an intermediate representation (IR), many of which are machine independent. Most compilers convert C code to IR before generating assembly. Program-analysis tools usually use a front-end to translate the program to some IR suitable for analysis and optimizations. Such tools also contain a back-end to convert IR back to C.

The output is often generated in a canonical form, more limited than general C syntax (for example, SUIF, SimpleC).

## 5.1 IR-C

Similar to *CoreC*, IR-C (see [Leu02]) retains the executability of the output by representing the IR in C syntax. Another similarity is the attempt to simplify C expressions into three-address instructions. IR-C produces lower-level code than *CoreC*.

## 5.2 CIL

The CIL (C Intermediate Language) in [NMRW02] is a high-level intermediate representation for C which has many features common to *CoreC*. However, *CoreC* conceptually defers from CIL. In general, *CoreC* is more structured and performs deeper simplification of the input program. The main differences between *CoreC* and CIL are listed in Table 4.

<i>CoreC</i>	CIL
Converts initialization of a variable in its declaration to a regular assignment, which requires removing <i>const</i> modifiers from all types. This conversion is applied to global variables as well as locals. Therefore, initialization of global variables is performed in a meta-main function implanted in the beginning of the original main function.	Applies the same conversion only for local variables, whereas global variables can be initialized at declaration.
Converts initialization lists and array initializations to assignments, including assignments to missing elements.	Applies these conversions only for local variables.
Always removes operators $? \rightarrow []$ .	Index operator is removed only for pointers, not for arrays.
Converts static variables in functions into global variables.	—
—	Adds missing function declarations.
Supports only <i>goto</i> -statement and <i>if</i> -statement with <i>goto</i> 's on its branches.	Supports additional control flow constructs: <i>while</i> , <i>break</i> , <i>continue</i> , <i>switch</i> .
Transforms expressions with side-effects into sequences of statements.	Retains ++ operator.
Supplies original code line	—

Table 4: Differences between the features supported by CIL and *CoreC*.

<i>CoreC</i>	CIL
Does not allow nesting, except for fields.	Allows nested expressions (both fields and operators), which reduces the number of temporary variables and new statements introduced by the translation.
Retains scopes.	Removes scopes inside functions (requires renaming)
—	Removes declarations for unused entities (types, variables).
Always converts expressions with <code>&amp;&amp;</code> , <code>  </code> into explicit if-statements.	Treatment of boolean expressions and conditions: (i) convert expression with <code>&amp;&amp;</code> , <code>  </code> operators into explicit if-statement. (ii) duplicate expressions and statements that depend on boolean operation. (iii) to avoid excessive duplication use a <i>goto</i> for statements that have more than 5 instructions.
Has only a canonic form of if-statement: <code>if () goto L1 else goto L2</code> .	Allows arbitrary statements as if-clauses.
Implemented under Windows with Microsoft Visual Studio compiler.	Implemented for UNIX systems with GCC and has support for special GCC options.

Table 4: Differences between the features supported by CIL and *CoreC*.

## 6 Complexity

The number of new variables introduced during the translation process is linear in the number of operations in the input program. For each new variable, exactly one assignment is performed. The only exception is in the translation of `&&` and `||` operators to an expression, where the new variable that denotes the value of the expression is assigned in both branches of an if-statement. The number of new labels is linear in the number of loop (while, do-while, for) and conditional (if, case) constructs in the program. The translation of initialization lists and array initializations is linear in the number of initializers.

## References

- [App98] A. W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.
- [Deg95] J. Degener. ANSI C yacc grammar. <http://www.lysator.liu.se/c/ANSI-C-grammar-1.html>, 1995.

- [DRS03] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *To appear in SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 2003.
- [KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.
- [Leu02] R. Leupers. An executable intermediate representation for re-targetable compilation and high-level code optimization. Unpublished, 2002.
- [NMRW02] G. Necula, S. McPeak, S. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Int. Conf. on Comp. Construct.*, pages 213–228, 2002. See “<http://manju.cs.berkeley.edu/cil/>”.
- [TSBTG] Microsoft Research The Semantics Based Tools Group. Microsoft ast toolkit. Licensed Software.
- [Yor02] G. Yorsh. Translation rules from C to CoreC - informal pseudo-code description. Available at “<http://www.math.tau.ac.il/~grey/GFC>”, 2002.