A Type-checking Preprocessor for Cilk 2, a Multithreaded C Language

by

Robert C. Miller

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 1995

Copyright 1995 Massachusetts Institute of Technology. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
May 12, 1995

Certified by .....
Charles E. Leiserson
Thesis Supervisor

Accepted by .....
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

A Type-checking Preprocessor for Cilk 2, a Multithreaded C Language

by

Robert C. Miller

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1995, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the type-checking, optimizing translator that translates Cilk (a C extension language for multithreaded parallel programming) into C. The Cilk-to-C translator is based on C-to-C, a tool I developed that parses, type checks, analyzes, and regenerates a C program. With a translator based on C-to-C, developers and users of C extension languages can enjoy the benefits of a type-checking, optimizing compiler without the attendant development and porting costs. Like a compiler, the Cilk-to-C translator runs quickly, does static checking, and generates efficient code, making it fit for everyday use by ordinary programmers. Unlike a compiler, however, the Cilk-to-C translator is easy to develop, extend, and port to other platforms. About 90% of the development effort was devoted to Cilk semantics (translating Cilk into C), which is where a language developer wants to focus. Only a small amount of work was spent extending C-to-C's parsing, type checking, and data-flow analysis to recognize Cilk. In return for a few weeks of full-time effort, I obtained a type-checking, optimizing translator for Cilk that targets any platform with an ANSI C compiler. With C-to-C, other C extension language developers can obtain the same benefits.

C-to-C has a number of special features that make it a good framework for C extension language translators. First, its C output is human-readable, so that a language developer or programmer can read and debug it. C-to-C also provides operational transparency (reproducing nonsyntactic source features like line numbering and indentation), which contributes to the readability and portability of its C output and assists debuggers and profilers. C-to-C performs data-flow analysis directly on the high-level abstract syntax tree of the source program, so that its C output is also high-level, not unreadable "assembly language written in C." Finally, C-to-C provides a generic data-flow analysis abstraction, in which any monotonic data-flow problem needed for optimization can be specified and solved automatically. These features greatly simplify the task of writing a type-checking, optimizing translator for an arbitrary C extension language.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Contents

1 Introduction **7**
- 1.1 One alternative: a macro preprocessor . . . 8
- 1.2 Another alternative: a full compiler . . . 8
- 1.3 Related work . . . 9
- 1.4 Outline . . . 9

2 The Cilk Programming Language **11**
- 2.1 An example of Cilk: `fib` . . . 11
- 2.2 Type checking . . . 12
- 2.3 Type-directed translations . . . 13
- 2.4 Optimizations . . . 14

3 The C-to-C Type-checking Preprocessor **17**
- 3.1 The abstract syntax tree . . . 17
- 3.2 Phases . . . 19
  - 3.2.1 Macro preprocessing (`cpp`) . . . 19
  - 3.2.2 Parsing . . . 20
  - 3.2.3 Type checking . . . 20
  - 3.2.4 Analysis . . . 20
  - 3.2.5 Transform . . . 21
  - 3.2.6 Unparsing . . . 21

4 Transparency in C-to-C **23**
- 4.1 Pragma directives . . . 24
- 4.2 Line numbering and indentation . . . 25
- 4.3 Constant expressions . . . 26

5 Data-flow Analysis in C-to-C **29**
- 5.1 Representing control flow in the AST . . . 30
- 5.2 Data-flow analysis frameworks . . . 31
- 5.3 The iterative algorithm . . . 32

6 Conclusions **35**
- 6.1 Future work on C-to-C . . . 35
- 6.2 Getting C-to-C . . . 36

Acknowledgements

This research was supported in part by the Advanced Research Projects Agency under Grant N00014-94-1-0985.

I am indebted to my advisor Charles E. Leiserson. Yuli Zhou provided advice and support throughout the development of the Cilk preprocessor. Thanks to Laura Cassenti, Charles Leiserson, Anil Somayaji, and Yuli Zhou, who were generous with their time in agreeing to proofread this thesis; any glaring errors that remain are solely the responsibility of the author. Thanks also to the entire Cilk team for support and suggestions: Bobby Blumofe, Matteo Frigo, Chris Joerg, Bradley Kuszmaul, Irena Kveraga, Charles Leiserson, Howard Lu, Phil Lisiecki, Keith Randall, Richard Tauriello, Daricha Techopitayakul, and Yuli Zhou.

I gratefully acknowledge the warm support of Laura Cassenti and my parents, Larry and Marian Miller.

Chapter 1

Introduction

Cilk (pronounced "silk") is a parallel programming language under development at MIT Laboratory for Computer Science [5]. Cilk provides syntax for expressing control parallelism, allowing a programmer to specify that certain procedure calls should be *spawned*, or run in parallel with the current thread of execution. Cilk is an example of a *C extension language* — a programming language that extends C with new keywords, syntax, or semantics. C extension languages have lately become popular in parallel and distributed computing research [5, 7, 8, 9, 15], because of the wide portability of ANSI C, the large population of C programmers, and the extensive base of C applications and library software.

Like many C extension languages, Cilk is not translated directly into object code. Instead, Cilk is translated into C, with extension language features mapped into calls to a runtime system. Figure 1-1 shows the translation process schematically. The translator that converts a C extension language into C is called a *preprocessor*, because its output is a high-level language rather than machine language. The C *postsource* produced by the preprocessor is compiled into object code by the target machine's C compiler, also called the *back-end compiler*. This compilation system — a preprocessor pipelined with a compiler — yields portability and ease of development, because the extension language developer need not build a full compiler for every platform on which the language will run.

In fact, simple C extension languages can be translated into C by local, syntactic transformations, using a macro preprocessor. Cilk 1, the first incarnation of the Cilk language, was translated by a macro preprocessor. Though easy to write and debug, macro preprocessors suffer a serious flaw, which manifested itself in Cilk 1. A macro preprocessor relies on the back-end C compiler's type checking to detect and report common programmer errors. Unfortunately, type errors in Cilk 1 programs are often missed by the back-end compiler, because the C postsource contains low-level code that overrides type checking. Even when type errors are detected, the C compiler's error messages refer to the postsource, which is

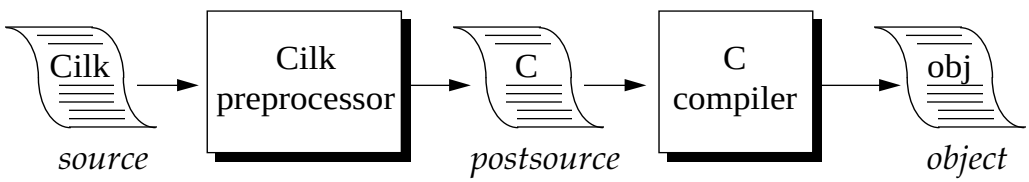


Figure 1-1: The mechanism for compiling a Cilk program. Cilk is preprocessed into C, which is then compiled into object code.

unhelpful to a Cilk programmer trying to find and fix the error in the source.

In addition, macro preprocessors are insufficient for more abstract C extension languages that require their translators to gather semantic information about the program. Cilk 2, the latest version of the Cilk language, includes features whose translation depends on the types of the objects involved, so the Cilk 2 preprocessor must determine those types. An ordinary macro preprocessor has no access to type information.

Finally, macro preprocessors offer no opportunity for global optimization. The back-end C compiler performs global optimization when it generates object code, but the compiler's optimizations are necessarily conservative, missing potential optimizations that the preprocessor can perform. Again Cilk 2 serves as an example. Gathering data-flow information enables the Cilk 2 preprocessor to emit faster code than a macro preprocessor could.

This thesis studies type-checking, optimizing preprocessors for C extension languages, using the Cilk 2 preprocessor as an example. In the course of building the Cilk 2 preprocessor, I have developed a generic preprocessor framework for C extension languages called C-to-C. C-to-C parses a C program into an abstract syntax tree (AST) representation, performs type checking, data-flow analysis, and tree transformations directly on the AST, then unparses the AST to recover the original C program. To build a C extension language preprocessor based on C-to-C, a developer modifies the front-end to recognize the extension language, then adds transformations that convert extension syntax into ordinary C syntax. The Cilk 2 preprocessor was built in precisely this manner.

1.1 One alternative: a macro preprocessor

One goal of this thesis is to show that writing a type-checking, optimizing preprocessor based on C-to-C is a reasonable alternative to using a macro preprocessor. As the preceding discussion demonstrated, type checking and optimization increase the usability, abstraction level, and performance of Cilk, so the Cilk 2 type-checking preprocessor is superior to the Cilk 1 macro preprocessor. With C-to-C, these benefits were achieved at low cost. Our experience with the Cilk preprocessor suggests that extending C-to-C to build a type-checking preprocessor is only a little more difficult than writing macros. In fact, only a small fraction of the development time for the Cilk 2 preprocessor was spent extending C-to-C's type checking and data-flow analysis. Most of the development time was devoted to translating Cilk into C — essentially the same work that would be required for a macro preprocessor. C-to-C translations are somewhat harder to write than macros, since they involve tree manipulations instead of textual substitutions, but I believe that the extra effort is more than justified by the rewards.

1.2 Another alternative: a full compiler

The second goal of this thesis is to argue that writing a preprocessor based on C-to-C is an attractive alternative to another common approach to translating experimental C extension languages – extending a full compiler. This approach begins with a standard C compiler and extends its front end to recognize the new extensions [2, 16]. Compared to a full compiler, however, a translator based on C-to-C is portable and extensible — highly desirable qualities for language research. In addition, the output of a C-to-C translator is readable, high-level C, not assembly language, nor even "assembly language written in C" as might be expected from an optimizing translator. This section will examine these arguments in more depth.

First, a translator based on C-to-C is easier to port than a full compiler. C-to-C itself is written in portable ANSI C, and extension language features are translated into high-level C, which can be made portable. For example, the Cilk preprocessor generates portable C containing calls to a runtime system, so moving Cilk to a new architecture entails no changes to the preprocessor whatsoever. Only the relatively small runtime system must be ported.

Next, a translator based on C-to-C is easier to extend than existing public-domain C compilers. Its lexical scanner and parser are automatically generated from `lex/yacc` specifications, which are easier to change than the hand-coded front-ends of `lcc` [10] and the SUIF compiler framework [2]. C-to-C's type checking and data-flow analysis are table-driven, so that new statements and expressions can be specified by just a few method functions. Transformations are expressed as operations directly on the high-level abstract syntax tree, which we found easier to manipulate than the intermediate representation of compilers like `gcc` [16]. Also, unlike most compilers, C-to-C provides an abstraction for data-flow analysis which can be extended to solve any monotonic data-flow analysis problem needed for translation or optimization.

Finally, a translator based on C-to-C produces output that is high-level, readable, and readily compared with its input, since only extension language features are translated. C-to-C provides "operational transparency," preserving all high-level C syntax, line numbering, indentation, and constant expressions used in the original program. The readability of C-to-C's output simplifies the process of checking that extension language constructs have been translated correctly.

1.3 Related work

The Sage++ preprocessor framework [6] is similar to C-to-C in that it provides a rich toolkit for program analysis and transformation, including source transparency and data-flow analysis. Although the Sage++ front-end supports C, C++, and Fortran, it is more difficult to extend to new languages. Its lexical scanner is hand-coded, and its parser and type-checker use optimized, low-level interfaces to the AST rather than the high-level C++ classes. Also, Sage++ is written in C++, which is less widespread than ANSI C.

1.4 Outline

The remainder of this thesis is organized as follows. Chapter 2 describes the Cilk 2 language and explains why it requires type checking and data-flow analysis from its preprocessor. Chapter 3 introduces the C-to-C preprocessor framework and describes its architecture. Chapter 4 explains how C-to-C preserves nonsyntactic information, like line numbering and indentation, from the source to the postsource. Chapter 5 describes how C-to-C performs data-flow analysis directly on the source program. Chapter 6 offers some conclusions contrasting C-to-C-based preprocessors with other alternatives for translating C extension languages, and describes our plans for future work.

Chapter 2

The Cilk Programming Language

This chapter describes the parallel C extension language Cilk. The discussion below is neither a complete specification of the language nor a detailed description of how Cilk extensions are translated into C. That information is available elsewhere [4]. Rather, the first section of this chapter will present an example of Cilk and describe its special features, and the remaining sections will show how those features are translated into C, in order to explain why Cilk requires type checking and data-flow analysis from its preprocessor.

2.1 An example of Cilk: `fib`

Cilk programs are run in parallel on multiple processors. The basic unit of parallel computation in a Cilk program is a *procedure*, which is similar to a C function except that it can run in parallel with other Cilk procedures.

The best way to explain Cilk is by example. The Cilk procedure `fib` shown in Figure 2-1 computes Fibonacci numbers recursively, with the recursive calls proceeding in parallel. Most of the code is ordinary C, which is passed through the Cilk preprocessor unchanged, except for the statements and declarations containing keywords in boldface, which must be translated into C. These keywords are briefly explained below:

- **procedure** identifies `fib` as a Cilk procedure. Only Cilk procedures can be *spawned*, or called in parallel.

```
procedure int fib(int n)
{
  if (n < 2)
    return n;
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return x+y;
  }
}
```

Figure 2-1: The Fibonacci function computed by a recursive Cilk procedure. Features in boldface must be translated by the Cilk-to-C preprocessor.

- **spawn** calls a Cilk procedure in parallel with the current thread of execution. The procedure initiating the `spawn` is the *parent*, and the procedure spawned is its *child*. A spawn statement must assign the child's return value to a local variable, whose value is undefined until the parent executes a `sync` statement to wait for the child to return.

- **sync** waits for all children started with `spawn` to return. After a `sync` statement, the return values of all previously-spawned children are computed and available.

- **return** returns a value to the procedure's parent. A Cilk procedure's `return` statement differs from the ordinary C `return` statement, since the Cilk procedure may need to communicate the return value to a different processor.

Cilk includes other features that provide the programmer with greater control over synchronization and scheduling of parallel procedures, which are described elsewhere [4].

The remaining sections of this chapter will use the `fib` example to illustrate the following claims. First, the back-end C compiler's type checking fails to detect some static type errors in Cilk programs and reports unhelpful error messages for others, so the preprocessor must perform Cilk-specific type checking. Second, some Cilk extensions require type-directed translations, in which the C translation depends on the types involved. Thus, the preprocessor must determine and store type information. Third and finally, some Cilk extensions cannot be optimized by the back-end C compiler, so the preprocessor needs global flow analysis to generate optimized C code.

2.2 Type checking

This section shows why the type checking provided by the back-end C compiler is insufficient for Cilk, considering as an example the `spawn` statement. A type-incorrect `spawn` statement translates into type-correct C code, so the type error cannot be detected by the back-end C compiler. As a result, the Cilk preprocessor must perform type checking directly on the Cilk source.

The C code corresponding to one of the `spawn` statements in the Fibonacci example is shown in Figure 2-2. This fragment of C code calls the runtime system function `NewFrame` to construct a *frame*, which is a migratable activation record for the procedure being spawned. A frame is a C structure containing slots for the procedure's formal arguments and local variables, along with some housekeeping information required by the runtime system. The frame is initialized with the procedure's single argument `n` and a special pointer called a *continuation*. The continuation points to the slot in the parent procedure's frame which

```
x = spawn fib(n-1);     ⇨     {
                                struct fib_frame *new_fp;
                                newfp = NewFrame(fib, fib_fsize);
                                newfp->n = n-1;
                                newfp->ret = MakeContinuation(fp, offsetof(fp, x));
                                PostFrame(newfp);
                              }
```

Figure 2-2: Translation of Cilk `spawn` statement into C (details omitted for clarity).

should receive the child's return value – in this case, the slot representing the local variable `x`. The continuation is constructed from a pointer to the parent procedure's frame structure and the byte offset of `x` in that structure. Finally, `PostFrame` places the new frame in the scheduler's work queue. It may be executed later on the current processor or migrated to another processor that needs work to do.

Observe that the C code deals with `x` on a low level, using its byte offset in the frame to construct a continuation. Thus, if the programmer accidentally assigns a spawn to a local variable of incorrect type, the back-end C compiler will fail to detect the type error. As a result, a mistake that was statically detectable can slip through to runtime, resulting in a runtime error, or worse, an incorrect answer.

2.3 Type-directed translations

This section gives an example of a type-directed translation to show why the Cilk preprocessor must collect type information from the Cilk source. The example is the Cilk `return` statement, whose translation into C depends on the return type of the procedure in which it appears. The `return` statement is also used to illustrate that, even when the back-end C compiler succeeds in detecting a type error in a Cilk program, it often reports an unhelpful, confusing error message. This example reaffirms the need for type checking in the Cilk preprocessor.

The C translation of one of `fib`'s `return` statements is shown in Figure 2-3. A Cilk procedure returns a value to its parent using one of a family of runtime system primitives `Send`*Type*`Argument`, where *Type* can be any C arithmetic type (`Char`, `Int`, `Float`, `Double`, etc.). In order to choose the appropriate runtime system primitive to replace the `return` statement, the preprocessor must determine the return type of the enclosing procedure.

The `return` statement also illustrates how relying on the back-end C compiler for type checking can result in unhelpful error messages, since the messages describe the C postsource rather than the original Cilk source. If the expression in the `return` statement is incompatible with the return type of the procedure, then the back-end C compiler reports an error message like "`type mismatch in argument 1 of call to SendIntArgument`." This message makes no reference to the fact that the real problem was a type error in the Cilk `return` statement. Admittedly, even the simplest Cilk preprocessor can cause the line number of the `SendIntArgument` call in the postsource to correspond with the line number of the `return` statement in the source, so at least the error message correctly identifies the offending line in the source. But a Cilk programmer unfamiliar with the output of the preprocessor is likely to be confused by the reference to `SendIntArgument`, and may have difficulty figuring out what is wrong with the offending line and how the mistake should be

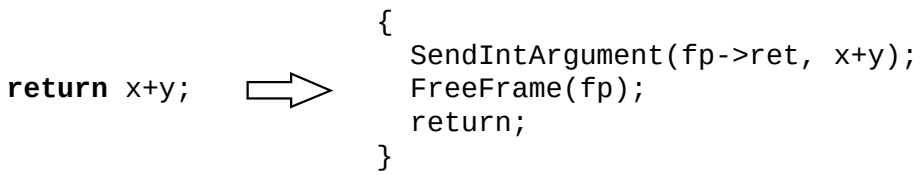
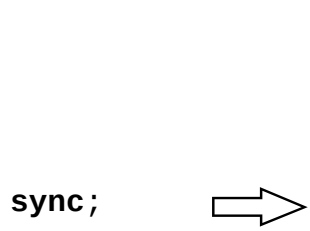


Figure 2-3: Translation of a Cilk `return` statement into C. This statement appeared in the `fib` procedure of Figure 2-1.



```
sync;
```

```
{
  fp->x = x;
  fp->y = y;
  fp->n = n;      /* UNNEEDED! */
  fp->entry = 1;
  return;
sync1:
  x = fp->x;
  y = fp->y;
  n = fp->n;      /* UNNEEDED! */
}
```

Figure 2-4: Unoptimized translation of Cilk `sync` statement into C (some details omitted for clarity). This `sync` appeared in the `fib` procedure shown in Figure 2-1. In that procedure, `n` is not used after the `sync`, so it does not need to be saved and restored.

corrected.

2.4 Optimizations

This section shows why data-flow analysis is important to the Cilk preprocessor, using the `sync` statement as an example. Since the back-end C compiler must be conservative in its optimizations, it misses an important optimization opportunity in the translation of the `sync` statement. With data-flow analysis, the Cilk preprocessor can perform the optimization itself.

A naive (unoptimized) translation of the `sync` statement in `fib` is shown in Figure 2-4. The `sync` statement suspends `fib` until its spawned children have returned. To wait for its children, `fib` must return control to the scheduler, which it does with an ordinary C return statement. The scheduler works on other parallel procedures until the children return, at which point the scheduler calls `fib` in such a way that control resumes at the label `sync1`. In order to preserve its state of execution, `fib` saves its local variables before returning to the scheduler, and restores them after resuming.

The C code shown in Figure 2-4 is not efficient, because it saves and restores *all* local variables. In particular, it unnecessarily saves and restores the formal parameter `n`. A glance at Figure 2-1 confirms that `n` will not be used after the `sync`, so its value need not be preserved.

Normally, the Cilk preprocessor is not concerned with generating carefully optimized C code, as long as it can rely on the back-end C compiler to generate optimized object code. In fact, an optimizing C compiler *can* remove the restoring assignment `n = fp->n`, because the local variable `n` is not used subsequently. But no C compiler in existence can optimize out the saving assignment `fp->n = n`. A general-purpose compiler cannot assume that the frame slot `fp->n` will not be needed later, because the frame is stored in ordinary memory, possibly aliased by other pointers in runtime system data structures, and manipulated at the byte level by the runtime system. The frame can even be migrated from processor to processor, in which case all bets are off as far as the C compiler is concerned.

The Cilk preprocessor, on the other hand, knows that the runtime system does not use

`fp->n` or any local variable slot in the frame. Only the Cilk procedure's code reads or writes those slots. Knowledge of this invariant about Cilk allows the preprocessor to remove the saving assignment as well, an optimization that is impossible for the C compiler.

In order to perform the optimization, the preprocessor must use data-flow analysis to determine which variables may be *live*, or needed later, and emit C code that saves and restores only those variables.

Chapter 3

The C-to-C Type-checking Preprocessor

The preceding chapter motivated the need for type checking and data-flow analysis in the Cilk preprocessor. To partition the development process and increase the potential for code reuse, I first built a model preprocessor called C-to-C that translates C into C, and then I extended it to recognize and translate Cilk. This chapter describes the architecture of C-to-C, focusing on how it can be extended to build a type-checking preprocesor for an arbitrary C extension language.

C-to-C encapsulates the organization of a generic type-checking preprocessor. It consists of a sequence of phases operating on a shared data structure, called an abstract syntax tree (AST). Each phase annotates or transforms the AST. The AST is constructed from a C program by a parser, annotated by type checking and data-flow analysis, transformed by tree operations, and finally unparsed to produce the original C program. To build a C extension language preprocessor based on C-to-C, a developer modifies the parser, type-checker, and data-flow analyzer to recognize the extension language, then adds transformations that convert extensions into C. The Cilk preprocessor, "Cilk-to-C," was built in precisely this manner.

C-to-C is based on `c-parser`, an ANSI C static checker written by Eric A. Brewer and Michael D. Noakes at the MIT Laboratory for Computer Science, with a `yacc` grammar for ANSI C written by James A. Roskind. This package is designed to parse a C source file, convert it into an abstract syntax tree, and perform static type checking on the tree. C-to-C's parsing and type checking phases and the structure of its abstract syntax tree are inherited from `c-parser`.

The first section of this chapter describes the structure of the abstract syntax tree. The remaining sections describe the phases of translation in C-to-C.

3.1 The abstract syntax tree

This section describes the abstract syntax tree, explains why it is a good representation for extension language preprocessing, and points out the special features of C-to-C's abstract syntax tree that enable type checking, data-flow analysis, and type-directed translation.

An abstract syntax tree (AST) represents the syntactic structure of a computer program in a tree data structure [1]. Each tree node in an AST represents a declaration, statement, or expression operator. Its children are the operands or syntactic components

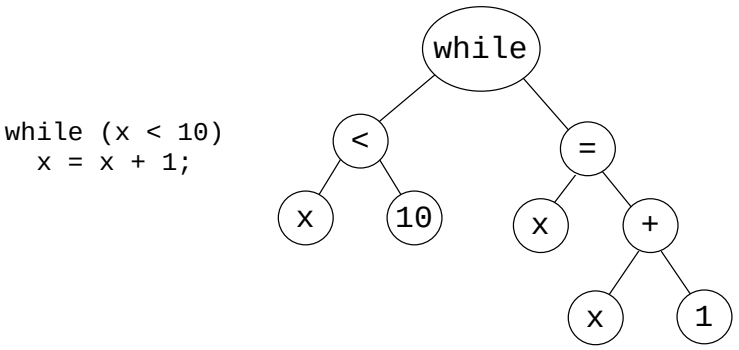


Figure 3-1: An abstract syntax tree (AST) for a C program fragment.

of the declaration, statement, or expression. Some examples of AST nodes are shown in Figure 3-1.

Abstract syntax trees are an ideal representation for syntax-directed macro preprocessors. The AST allows a preprocessor to perform translations locally, converting an extension statement or expression node into C while treating its operands as black boxes. For instance, the Cilk 1 macro preprocessor could transform a `spawn` statement, which is represented in the AST by a node with one child for each argument of the spawned procedure, by generating a replacement tree for the `spawn` with the argument expressions substituted (see Figure 3-2).

Since the AST representation is so convenient for C extension language translation, C-to-C also uses an AST, but extends the representation to include additional information. In order to perform type checking and semantics-directed translation, C-to-C annotates the AST with use-declaration pointers that point from identifier uses to their declarations, control flow pointers that identify the destination of nonlocal jumps like `goto`, variable and expression types, constant expression values, and data-flow analysis results. These semantic annotations are used to transform the source AST into an ordinary C AST.

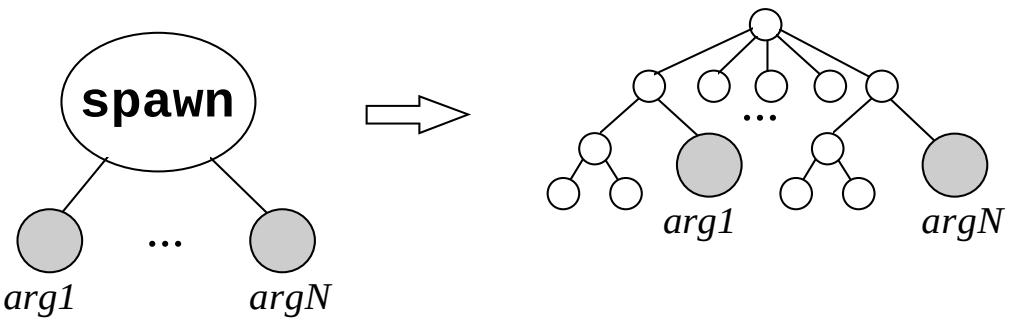


Figure 3-2: Translating a Cilk `spawn` statement into C by operations on the abstract syntax tree. The `spawn` arguments, shown here as gray circles, are substituted directly into the tree of C code that replaces the `spawn` node.

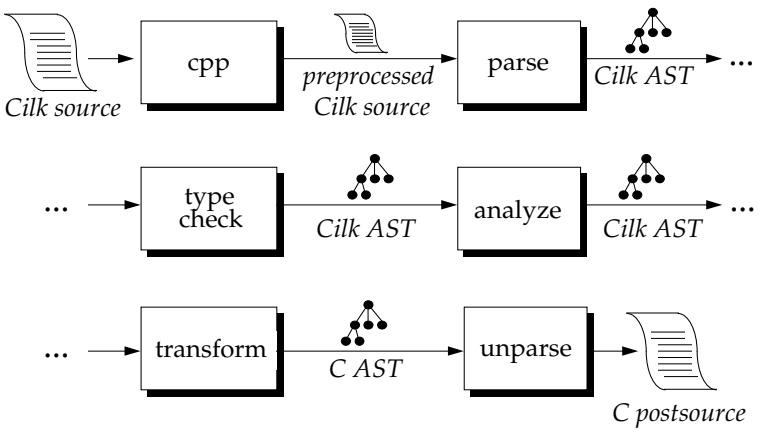


Figure 3-3: Translation phases of the Cilk preprocessor based on C-to-C.

3.2 Phases

This section describes the phases of translation in the C-to-C architecture, shown schematically in Figure 3-3. Actually, the figure shows the Cilk preprocessor based on C-to-C, in order to clearly distinguish the front-end phases, which operate on a C extension language like Cilk, from the back-end phases, which operate on C. (In the unextended version of C-to-C, the front-end and back-end languages are both C, so it would be hard to find the dividing line.) The purpose of each phase is summarized below. The remaining sections of this chapter will describe each phase of C-to-C in more detail, using the Cilk preprocessor to illustrate how to extend each phase.

Cpp performs standard C macro preprocessing on the source file, producing a file with no preprocessor directives or comments.

Parse parses the source program into an AST.

Type check performs type checking on the AST. annotating every declaration and expression node with type information.

Analyze performs data-flow analysis on the AST, annotating every statement and expression node with data-flow information.

Transform performs tree transformations that convert the annotated source-language AST into a pure C AST. (In C-to-C, this phase does nothing, because the source language and target language are both C.)

Unparse unparses the AST into text.

3.2.1 Macro preprocessing (`cpp`)

The `cpp` phase performs standard C macro preprocessing [13]. During this phase, macro definitions are interpreted and expanded, conditional compilation directives are recognized, and comments are removed from the source. The output of preprocessing is a file of text with no preprocessor directives or comments.

The current implementation of C-to-C invokes the back-end compiler to perform C macro preprocessing. One consequence of this decision is that the `cpp` phase cannot be extended. It also limits the portability of C-to-C: since ANSI did not standardize the connection between the C macro preprocessor and the C compiler, some back-end compilers may not offer a way to invoke their macro preprocessing pass by itself. Even a compiler that allows separate invocation of its macro preprocessor may emit private, non-standard compiler directives in the preprocessor output. To remedy these and other deficiencies, a future implementation of C-to-C may include its own `cpp` macro preprocessor.

3.2.2 Parsing

The parsing phase consists of a lexical analyzer (generated automatically by `lex`) and an LR(1) parser (generated automatically from an LR(1) grammar by `yacc`) that together parse the source program and construct an AST representing it. If any syntax errors occur in the source, they are reported by the parsing phase. The parsing phase also connects identifier uses to their declarations by use-declaration pointers, and nonlocal jumps to their destinations by control-flow pointers. The result of parsing is an AST annotated with declaration and control-flow pointers.

To extend the parsing phase to recognize Cilk, the Cilk preprocessor adds new tokens, described by regular expressions, to the lexical analyzer, and new syntax, described by context-free productions, to the grammar.

The parsing phase is based on the Brewer-Noakes ANSI C static checker. The `lex/yacc` parser is based on the ANSI C parser by James A. Roskind.

3.2.3 Type checking

The type checking phase makes a pass over the AST, determining types for expressions and computing values for constant expressions. During the same pass, it looks for type mismatches and makes other static semantic checks required by ANSI C, and reports any errors to the user. The result of type checking is an AST annotated with types and constant values.

Type checking is driven by a table of method functions, one for each kind of AST node. The checks required for a new statement or expression can be defined by adding a method to the table. The Cilk preprocessor adds methods that type-check its unique statements, like `spawn` and `sync`. In a C extension language that defines new types, like the Cilk `procedure` type, it may also be necessary to modify the type-checking methods of ordinary C statements and expressions. For instance, Cilk allows a Cilk `procedure` to be called like a C function. In order to allow this usage, the Cilk preprocessor extends the type-checking of an ordinary C function call to accept a Cilk `procedure` as well a C function in the operator position.

The type checking phase is also based originally on the Brewer-Noakes static checker, though significant changes have been made.

3.2.4 Analysis

The analysis phase performs intraprocedural data-flow analysis on every procedure in the AST. The analysis phase can compute the solution of any data-flow problem that can be represented in a monotonic data-flow analysis framework, including live variables, reaching definitions, and constant propagation. The algorithm is iterative, making repeated passes

over a procedure until the data-flow equations converge to a solution. Monotonic frameworks and the iterative algorithm are discussed in detail in Chapter 5.

The analysis phase also makes some semantic checks, warning the user about unreachable statements and functions which do not return a value. The result of data-flow analysis is an AST annotated with data-flow analysis information, such as live variables at every statement.

Like type checking, data-flow analysis is also table-driven. In the Cilk preprocessor, the control flow of Cilk statements was defined by adding methods to the table. In addition, a developer can extend the analysis to solve additional data-flow problems, by defining the problems as monotonic frameworks. The Cilk preprocessor, for instance, defines two data-flow problems: live variable analysis, which determines whether a variable may be used before its next assignment, and "dirty variable" analysis, which determines whether a variable has been assigned since it was last saved to the frame.

3.2.5 Transform

The transform phase makes zero or more passes over the AST to convert C extensions into pure C. In C-to-C, this phase does nothing, since no C extensions are recognized by the front-end. In the Cilk preprocessor, however, the transform phase converts the Cilk syntax tree into a C syntax tree. The output of the transform phase is a C AST.

To an extension language developer, the transform phase is the most important and most interesting phase, since it specifies how extensions are translated into C. The transform phase accounted for more than 60% of the new code needed to extend C-to-C into the Cilk preprocessor, and about 90% of the development time.

3.2.6 Unparsing

The last phase, the unparsing phase, walks over the C AST and emits C code corresponding to it. The unparsing phase is designed to be the inverse of the parsing phase, so that re-parsing the output with C-to-C produces an identical AST.

Since the Cilk preprocessor generates ANSI C, it does not need to extend the unparsing phase. Other extension language preprocessors might extend the unparsing phase, if the target language is a language other than ANSI C.

Chapter 4

Transparency in C-to-C

This chapter describes the special care taken by C-to-C to imitate one of the strengths of simple macro preprocessors: transparency. We say a translator is *transparent* to a feature of the source program if the translator copies the feature unchanged from input to output, preserving it through any intermediate program representations. Transparency is good for three reasons. First, a transparent translator is more portable, because it avoids making changes that could produce unportable code. Second, a transparent translator interacts well with other programming tools that rely on nonsyntactic features of the source, like line-oriented debuggers and profilers. Finally, transparent translators are easier to debug, because the postsource is high-level and readable, and extension translations appear in context in the postsource. Text-based macro preprocessors are transparent to all features, since they deal directly with the text of the program. C-to-C, however, converts the program text to and from an abstract syntax tree representation, which is not ideal for representing all features of the source. Nevertheless, our implementation of C-to-C is transparent to `#pragma` directives, line numbering, indentation, and constant expressions.

On a high-level, C-to-C converts a C program (the *source*) into an AST, then converts the AST back to a C program (the *postsource*). In principle, the postsource should be identical to the source — a goal we call *complete transparency*. Unfortunately, the current implementation of C-to-C cannot satisfy complete transparency, because an external macro preprocessor removes comments and macros from the source before it reaches C-to-C.

In practice, however, the back-end C compiler is the only consumer of the postsource, so not all information in the source is important. Whitespace and comments, intended for human users, are ignored by the back-end compiler, and macros and preprocessor directives can be interpreted early without affecting the final object code. For the purpose of extension language preprocessing, it is unnecessary to capture and maintain all this extra information in the AST.

Thus, C-to-C meets a weaker specification, which we call *operational transparency*: if the source is a C program with no syntactic or semantic errors, then the postsource is a C program with no errors that is compiled to identical object code by the back-end compiler. If the input contains errors, then C-to-C should issue appropriate error messages without emitting the postsource. Happily, this specification is as easy to test as complete transparency: we can compile the source and the postsource separately, then perform a byte-by-byte comparison of their object code files.

Operational transparency gives more freedom to an implementor of C-to-C, but it should not give so much freedom that C-to-C is useless as a preprocessor framework. C-to-C should

still be transparent to features of the source that do not affect object code, such as line numbering and high-level syntax. In particular, we note the following desiderata for a good implementation of C-to-C:

- The postsource should be *portable*. In particular, if the source is a strictly conforming ANSI C program, then the postsource should also be a strictly conforming ANSI C program [3]. Thus, C-to-C should not perform transformations that depend on a particular target platform or back-end C compiler. An example of an unacceptable transformation is constant expression folding, which will be discussed further below. Portability allows a C-to-C-based preprocessor to target any platform with a standard ANSI C compiler.

- The postsource should have the same *line numbering* as the source, so that debuggers and profilers that rely on matching line numbers in the object code to line numbers in the source can function correctly.

- The postsource should be *high-level*. C-to-C should not, for instance, transform `while` loops into labels and `goto` statements, nor should it flatten nested expressions into sequences of simple operations with temporaries. Preserving the high-level features of the source contributes to the readability of the postsource, and allows an extension language developer to view extension translations in the context of the original source.

- The postsource should be *formatted for reading*, since human programmers have to read and debug it while developing a C-to-C-based preprocessor. Rather than emit all the postsource on one incredibly long line, for instance, C-to-C follows the line breaks and indentation style of the source as closely as possible.

The remainder of this chapter examines how these transparency requirements affect the manner in which C-to-C handles `#pragma` directives, line numbering and indentation, and constant expressions, all of which must be preserved in the AST. The next chapter will explain how transparency requirements affect data-flow analysis.

4.1 Pragma directives

The ANSI standard permits a compiler to recognize private compiler directives preceded by `#pragma`. This section describes how C-to-C preserves unrecognized `#pragma` directives from the source to the postsource, so that the source can include compiler directives intended for the back-end C compiler.

C-to-C itself recognizes two `#pragma` directives: `#pragma lang +C`, which puts the lexical analyzer into a special C-only mode, and `#pragma lang -C` which ends the special mode. Of course, these directives have no effect in the unextended version of C-to-C, since C-to-C recognizes only C anyway. They are provided in anticipation that a C extension preprocessor based on C-to-C would need a way to turn off its special reserved words and treat part of the input program as strict C. Many Cilk programs, for example, are mixes of existing C code and new Cilk code. With these `#pragma` directives, the Cilk preprocessor can accept existing C code that happens to use Cilk reserved words as ordinary identifiers, without forcing all such identifiers to be manually renamed.

Other `#pragma` directives in the source are assumed to be intended for the back-end compiler, and C-to-C attempts to preserve them. Unfortunately, since `#pragma` directives

stmt **::= if (** *expr* **)** *stmt* **else** *stmt*

Figure 4-1: A grammar production for the `if-else` statement in C.

are like preprocessor directives, they are not constrained by the ANSI C grammar and may appear between any pair of tokens in the input program. As a result, they cannot be expressed in the grammar. C-to-C handles this problem by collecting `#pragma` directives in the lexical analyzer. Then, before starting a new statement or declaration, the parser inserts any waiting directives into the AST. The drawback of this simple scheme is that `#pragma` directives that appeared within a statement or declaration are pushed to the end of the statement or declaration, possibly changing their meaning. (C-to-C generates a warning message when a pragma directive must be moved in this manner.) In our experience, however, these directives are generally used at the file level or statement level, and rarely appear in the middle of a statement or declaration.

4.2 Line numbering and indentation

C-to-C preserves line numbering and indentation from the source to the postsource by storing line numbers and character offsets in the AST. Preserving line numbering enables line-oriented debuggers and profilers to match object code with original source lines, and preserving indentation makes the postsource easier to read.

When it constructs the AST, the parsing phase annotates each node of the tree with the source file, line number, and character offset within the line where the syntax construct appeared. Such a triple $\langle file, line, offset \rangle$ is referred to as the node's *source coordinates*. Source coordinates are useful not only for identifying the offending source line in C-to-C's error messages, but also for providing the output phase with enough information to reconstruct the line numbering and indentation of the source, for the benefit of debuggers, profilers, and human programmers using the postsource.

Line numbering and indentation are preserved by saving the coordinates of tokens in the input program. Every token in the input program is a terminal in some production in the grammar, which is represented by a node in the AST. Consider, for instance, the production in Figure 4.2, which shows the `if-else` statement. An `if` node in the AST actually represents four input tokens, shown in boldface in the figure: `if`, `(`, `)`, and `else`. We could reproduce the input perfectly if we stored the coordinates of every one of these tokens in the `if-else` node. Such precision is overkill, however, since our goal is not exact reproduction of the input program, but approximate reproduction (with identical meaning). In typical C formatting styles, the positions of semicolons, commas, and parentheses are fixed relative to their neighbor tokens (always immediately following or immediately preceding), so C-to-C saves storage space in the AST by throwing away the coordinates of these tokens.

As a result, for most AST nodes, C-to-C saves the coordinates of only one token. For statement nodes, the coordinates of the statement keyword are saved. For operators, the coordinates of the operator token are saved. For two-keyword statements or operators, like `if-else` and `do-while`, the coordinates of both keywords are saved. For a block (a list of statements enclosed in curly braces), the coordinates of both curly braces are stored, since different programmers prefer their braces in different places.

Using the token coordinates stored in the AST by the parsing phase, the unparsing phase attempts to reconstruct the line numbering and indentation of the input. Since the

parsing phase didn't save coordinates for every input token, but only selected tokens, the reconstructed C code is not identical to the input, but it is close enough to meet the needs of line-oriented debugging and readability.

The transform phase of C-to-C may have added and deleted parts of the AST and moved other parts around. As a result, the unparsing phase cannot be guaranteed that the token coordinates in the AST are in their original order. Given a token T, its source coordinates $\langle file, line, offset \rangle$, and the coordinates of the current output position $\langle File, Line, Offset \rangle$, C-to-C must be able to emit T in an output position as close as possible to its source coordinates, which may have a completely different filename or line number. Fortunately, C provides a preprocessing directive that sets the filename and line number arbitrarily: `#line` *line* "*filename*". In order to prevent excessive `#line` directives from obscuring the readability of the postsource, C-to-C uses the following heuristic. A `#line` directive is emitted if and only if one of the following conditions is true:

- $file \neq File$, to change the output filename;
- $line < Line$, to back up to an earlier line in the output;
- $line > Line + \delta$, to advance a large distance in the output. The value δ is a small constant, typically 5. This heuristic compresses the large runs of empty lines usually left by the C macro preprocessor after removing comments and conditionally compiled code. With this heuristic, runs of more than δ empty lines are replaced by a `#line` directive jumping immediately to the end of the run.

After possibly emitting a `#line` directive, C-to-C inserts sufficient lines and spaces in the output to make $Line = line$ and $Offset \geq offset$. This simple algorithm suffices to match the line numbering and indentation of the original source.

4.3 Constant expressions

C-to-C preserves all constant expressions from the source to the postsource. For the sake of portability, it performs no constant folding, even though it must evaluate constant expressions for semantic checking. This section examines the problems with constant expressions evaluation more closely.

C-to-C must evaluate constant expressions to perform a variety of static semantic checks. For instance, in ANSI C, all array dimensions must be positive, the `case` expressions in a `switch` must have distinct constant values, and two array types are the same only if they have the same dimensions.

Evaluating constant expressions in C-to-C is complicated by the fact that many constant expressions in ANSI C are implementation-dependent: their values depend strongly on the back-end compiler and the target architecture. For example, `sizeof` expressions depend on the machine word size, and `sizeof(struct foo)` depends on how the back-end compiler chooses to arrange the `foo` structure in memory. Arithmetic on constant expressions can also return implementation-dependent results. One example is `~0` (the one's complement of 0), whose value depends on how many bits are in a machine word. For maximum portability to different back-end compilers, C-to-C should not make any assumptions about the values of such expressions.

Still, C-to-C cannot simply omit semantic checks involving implementation-dependent constant expressions, since the checks are important for discriminating strictly conforming

ANSI C programs from definitely erroneous inputs. For instance, all conforming ANSI C compilers accept `int foo[sizeof(int)]`, in which the array dimension is certainly positive, and reject (or warn about) `int foo[sizeof(int)-sizeof(int)]`, in which the array dimension is always zero. In order to conform to the ANSI standard, C-to-C must make the same discrimination.

C-to-C solves this problem by computing a value for every constant expression, using reasonable word sizes and structure packing rules, solely for the purpose of such semantic checks. The computed value is stored in the AST node of the expression for later use, but it does not replace the expression, nor is it used in the unparsing phase to perform constant-expression folding. Although C-to-C could conceivably recognize that an expression has a well-defined, implementation-independent value and fold the expression down to that value, this optimization is not worth the trouble, since the back-end C compiler is equally capable of constant folding. Furthermore, folding constant expressions reduces the abstraction level of the program, converting a high-level expression into a low-level expression. C-to-C delegates constant folding to the back-end compiler, where it belongs.

Chapter 5

Data-flow Analysis in C-to-C

This chapter describes C-to-C's data-flow analysis algorithm, which works directly on the high-level abstract syntax tree. C-to-C implements a standard iterative algorithm [1], but with performance improvements enabled by the AST representation, which include depth-first order and testing for convergence only at loops and jumps. It also provides an abstraction for monotonic data-flow analysis problems, so that an extension language developer can simply define the data-flow problem that needs to be solved, and C-to-C's data-flow analysis algorithm automatically solves it. The Cilk preprocessor uses this data-flow analysis framework to determine the live variables at a `sync` statement.

C-to-C performs data-flow analysis directly on the high-level AST, using an "implicit control flow graph" defined by the semantics of the C language. Because of the requirement of transparency described in the previous chapter, C-to-C cannot reduce the AST to a simpler intermediate language that contains no nested expressions. C-to-C cannot even treat a nested expression as a "basic block" of straight-line code, because many C expressions do not have straight-line control flow. Expressions can contain conditional operators or short-circuiting Boolean operators which may not evaluate their second operand. Worse, the GNU extensions to C allow arbitrary statements to appear inside expressions, so jumps into and out of the middle of an expression are possible [16]. (C-to-C supports this "statement expression" extension, because it is occasionally useful in Cilk programs.) To avoid these complications, C-to-C treats every AST node as a basic block, so its control flow graph (CFG) contains many more nodes than a typical compiler's control flow graph. Each iteration of C-to-C's data-flow analysis algorithm must propagate information through more nodes.

C-to-C offsets this performance hit by taking advantage of properties of the implicit CFG. First, the iterative algorithm converges fastest if the nodes of the CFG are visited in depth-first order on each iteration [11]. Since the AST is a depth-first spanning tree of its implicit CFG, we can visit CFG nodes in depth-first order by simply walking the tree, so fast convergence is automatic. Second, C-to-C tests for convergence only at nodes corresponding to jumps and labels, rather than at all nodes of the CFG as would a standard algorithm based directly on control-flow graphs.

The iterative algorithm coded in C-to-C applies to a variety of data-flow analysis problems, described in general as monotonic frameworks. Following the suggestions made by Kildall [14], C-to-C defines a monotonic framework abstraction which enables an extension language developer to define a data-flow analysis problem declaratively. The monotonic framework abstraction is described in detail below.

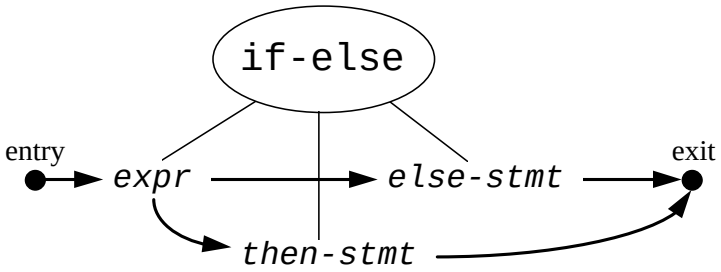


Figure 5-1: The implicit control flow of an `if-else` node. The node corresponds to the C code `if (`*expr*`)` *then-stmt* `else` *else-stmt*.

5.1 Representing control flow in the AST

C-to-C performs data-flow analysis on a control flow graph (CFG) represented implicitly in the AST. Representing the control flow graph implicitly rather than explicitly saves storage and makes C-to-C easier to extend.

Every statement or expression node in the AST has an implicit CFG. For example, in the `if-else` node shown in Figure 5.1, control passes first to the *expr* child, branches to either *then-stmt* or *else-stmt*, and then rejoins to leave the `if-else` node. The implicit control flow graphs for each AST node together specify an implicit control flow graph for the entire AST.

Most nodes of the AST receive control from the parent initially and return to the parent after execution. Some nodes, however, represent jumps or labels, such as `goto`, `break`, `continue`, `case`, and `switch`. The parsing phase annotates these nodes with control-flow pointers, connecting, for example, each `goto` node with its destination label node elsewhere in the AST. The implicit control flow graph of jump and label nodes makes use of these nonlocal links. The parser does not connect function calls with function definitions, however, and so C-to-C's flow analysis is presently limited to the scope of a single function. Intraprocedural analysis is sufficient for Cilk, and thus this restriction is not severe. Other C extension languages could extend C-to-C for interprocedural data-flow analysis, but the work required would be nontrivial.

Representing the CFG implicitly saves space. Most control flow follows existing AST edges, between parents and children in the tree, and an implicit CFG reuses the AST edges to save space. Only the jump and label statements mentioned in the previous paragraph have nonlocal control flow that does not follow AST edges, requiring additional space to store control-flow pointers.

The implicit CFG is also easy to extend, because it is table-driven. Each statement or expression node's implicit CFG is defined procedurally by a subroutine that passes data-flow information among the node's children, following the implicit CFG edges. The control flow of a C extension language can be defined simply by writing a new subroutine for each new statement or expression.

We should make one more observation about implicit control flow through expressions. C-to-C assumes that expressions and function call arguments are evaluated strictly from left to right, but the back-end compiler is free to choose any evaluation order when it generates actual object code. As a result, if the behavior of the source program depends on evaluation order, then C-to-C's analysis may be incorrect. Fortunately, programs that depend

upon evaluation order are rare and inherently unportable, since the order of evaluation of expressions and side-effects in C has always been undefined [3, 13]. Few programmers write code that depends on evaluation order, and those that do deserve what they get.

5.2 Data-flow analysis frameworks

This section describes C-to-C's data-flow analysis abstraction. For the sake of extensibility, C-to-C provides a "monotonic data-flow analysis framework" abstraction that is applicable to a wide variety of data-flow problems, including live variables, constant propagation, reaching definitions, and available expressions. Monotonic data-flow analysis frameworks were originally proposed by Kildall [14] and developed by Kam and Ullman [12]. C-to-C's implementation follows the treatment of monotonic frameworks in the "Dragon Book" ([1], section 10.11)

A monotonic data-flow analysis framework consists of:

Data type V: a set of values to be propagated. A value in V represents an assertion about the dynamic state of the program. The data type V must contain a distinguished element $\top$, which represents "undefined" or "unknown." The concrete data type used to implement V must provide an equality operation that tests whether two elements of V are the same, so that the iterative algorithm can detect convergence.

Direction of propagation: data-flow information may be propagated either forwards or backwards through the control flow graph.

Meet operation $\wedge$: a function mapping $V \times V \to V$, which combines assertions about two paths of execution where the paths join. The meet operation must be commutative, associative, idempotent, and has $\top$ as its identity element. With these properties, $(V, \wedge)$ is a semilattice. In particular, we can define the partial ordering $\leq$ on V by

$$x \leq y \iff x \wedge y = x$$

Thus, the distinguished undefined element $\top$ is the top of the semilattice, since $x \wedge \top = x$ for all $x \in V$.

Also, the semilattice $(V, \wedge)$ must be *bounded*: all chains $x_1 < x_2 < \cdots$ in V must have finite length. If the set of values V is finite, then this condition is automatically satisfied.

Transfer functions F: a set of functions mapping $V \to V$, which transform a value in V as it passes through an AST node. Two functions from F are assigned to each node: one to the entry point and one to the exit point. The set F generally should contain the identity function on V, since many AST nodes have no effect on data flow.

All functions in F must be *monotonic*:

$$f(x \wedge y) \leq f(x) \wedge f(y) \text{ for all } f \in F \text{ and } x, y \in V$$

The monotonicity of F and boundedness of $(V, \wedge)$ together imply that the iterative algorithm of successive approximation used by C-to-C eventually converges to a solution.

The functions in F should also satisfy *strictness*: $f(\top) = \top$ for all $f \in F$. Unlike most compilers, C-to-C does not remove obviously unreachable code from its AST before performing data-flow analysis. As a result, if F contains nonstrict functions, unreachable paths in the control flow graph could propagate information to the rest of the graph, despite the fact that they would never be visited at runtime. Thus, with nonstrict functions, C-to-C's data-flow analysis is too conservative. When all functions in F are strict, however, this problem does not arise. All unreachable paths remain $\top$ (undefined), which cannot affect the reachable paths of the program.

For an example of a data-flow analysis framework, consider the problem of computing live variables. A variable x is *live* at a point P in a program if x may be used before being reassigned on some possible path of computation starting at P. Live-variable analysis is useful to the Cilk preprocessor, because if a variable is dead (i.e., not live) at a `sync` point, then the preprocessor need not preserve its value across the `sync`.

Live-variable analysis fits into the framework as follows. The data type V is a set of variables, represented by a bit vector. A value of V asserts that the variables it contains are live at that point in the program. The direction of propagation is backwards, since liveness is a property determined by the future execution of the program. The meet operation $\wedge$ is set union, since a variable is live at a point in the program if and only if it is live on at least one path leaving that point. Finally, the transfer function set F contains two monotonic functions: $gen_x(S) = S \cup \{x\}$, which adds x to the set of live variables, and $kill_x(S) = S - \{x\}$, which removes x from the set of live variables. Every AST node representing a use of the variable x has gen_x as its exit transfer function, and every node representing an assignment to x has $kill_x$.

Monotonic data-flow analysis frameworks are important to C-to-C's extensibility. By defining a data type for V and writing subroutines for the meet and transfer operations, an extension language developer can reuse C-to-C's iterative algorithm to gather any data-flow information needed for translation or optimization.

5.3 The iterative algorithm

Given a framework and an AST, which together define a set of data-flow equations, C-to-C computes a solution to the data-flow equations using the standard iterative algorithm ([1], algorithm 10.18). The iterative algorithm begins with all edges in the implicit control flow graph set to $\top$, and iteratively visits nodes satisfying their data-flow equations until the solution converges to a fixed point. C-to-C's algorithm differs from the standard algorithm in two respects: the control flow graph is traversed in depth-first order, and convergence is only tested at loops and jumps.

Visiting CFG nodes in depth-first order yields the fastest convergence of the standard iterative algorithm [11]. If the data-flow problem is a distributive framework (or a monotonic framework satisfying certain assumptions), and if the CFG is reducible (each loop has at most one entry point), then the number of iterations before convergence is bounded by the *loop-connectedness* of the CFG, which is defined as the largest number of back edges on any cycle-free path of the graph. Intuitively, the loop-connectedness is the nesting level of the innermost loop in the graph. This result means that for most data-flow frameworks and well-structured C functions (which have only one entry per loop), the iterative algorithm requires only a few iterations, roughly as many as the depth of the innermost loop of the function.

To visit CFG nodes in depth-first order, other compilers must build a depth-first spanning tree on the CFG. In C-to-C, the AST is automatically a depth-first spanning tree of its implicit control flow graph, so C-to-C visits CFG nodes in depth-first order just by walking over the AST.

The standard iterative algorithm visits every node in every iteration, halting if and only if the data flow at all nodes remains constant across an iteration. Testing all nodes for changes during the iteration would be more expensive for C-to-C, because it does not summarize expressions and straight-line code into a single basic block.

C-to-C uses a faster termination condition that takes advantage of the high-level AST. C-to-C's algorithm halts if and only if the data flow at "loop nodes" and "confluence nodes" remains constant. A *loop node* is an AST node representing a loop in the source langage. In C, the loop nodes are `do-while`, `for`, and `while`. A *confluence node* is an AST node that may receive or pass control nonlocally, with another AST node that is neither its parent nor one of its children. The confluence nodes in C are `goto`, `switch`,[1] `continue`, `break`, `return`, label, `case`, and `default`. An *acyclic* node is neither a loop node nor a confluence node. The acyclic nodes in C are expressions and conditional (`if-else`) statements.

Because C-to-C visits nodes in depth-first execution order, the data-flow equations at acyclic nodes are always satisfied after every iteration. As a result, if the data flow at the loop nodes and confluence nodes remains unchanged across an iteration, then the data flow at the acyclic nodes also remains unchanged. Thus, it suffices only to test for convergence at the loop nodes and confluence nodes.

As a consequence of its termination condition, C-to-C performs only a single iteration if the AST contains no loop nodes or confluence nodes. An AST consisting entirely of acyclic nodes has an acyclic implicit control flow graph, so a single depth-first pass suffices to compute the solution to the data-flow equations.

[1] A `switch` statement in C is not as structured in C as it is in other languages. In C, `switch` is actually a computed jump to a label inside its body. Syntactically, the label is not a direct child of the `switch`, so we regard this jump as a nonlocal transfer of control.

Chapter 6

Conclusions

Using the C-to-C framework, I have written a preprocessor that translates both Cilk 1 and Cilk 2, performs type checking on the Cilk source, and uses live variable analysis to optimize saving and restoring local variables across `sync` statements.

Building the Cilk preprocessor on top of C-to-C required a relatively small amount of work, the equivalent of a few weeks of effort by a full-time programmer. In fact, about 90% of the programming effort was spent in figuring out how to transform Cilk extensions into C. Extending type checking and data-flow analysis to cover Cilk was comparatively simple. Our experience with the Cilk preprocessor suggests that C-to-C is a good preprocessor framework, since it allows an extension language developer to focus on the syntax and semantics of the extension language, rather than on type checking or data-flow analysis.

The type-checking preprocessor is now in everyday productive use by the Cilk group, who prefer it to the older Cilk 1 macro preprocessor. The type-checking preprocessor translates Cilk into C slower than the macro preprocessor but just as quickly as our C compiler (`gcc`) compiles C into object code. Overall, then, users of the type-checking preprocessor pay only a factor of two in compile time, in exchange for type-checking, optimization, and the ability to use Cilk 2 (a higher-level language than Cilk 1).

With an extensible framework like C-to-C, we can afford to contemplate more ambitious extensions to the Cilk language without cringing at the potential development effort. For instance, we are experimenting with a distributed shared memory system for Cilk, which is currently implemented with a low-level runtime library interface. To simplify programming the interface, we have devised a "global pointer" extension for Cilk which allows a programmer to manipulate shared objects with familiar C pointer operations. With the new Cilk preprocessor, we anticipate that adding global pointers to Cilk will take only a few days of work.

6.1 Future work on C-to-C

A future version of C-to-C may include its own standard C macro preprocessor. The current implementation relies on the back-end C compiler to perform macro preprocessing, which is neither portable (since not all C compilers offer a separate macro preprocessor) nor extensible.

Once C-to-C performs its own macro preprocessing, it will have access to the original source program, enabling a more transparent implementation of C-to-C. Enough information could be saved in the abstract syntax tree to reproduce the source program precisely,

including comments and macros.

Precise source reproduction would make C-to-C applicable to a wider variety of source-to-source processing applications. In C extension language preprocessing, the only consumer of C-to-C's output is a back-end compiler. But in many other applications of source-to-source translation, a human programmer might be another consumer, in which case preserving comments is extremely important. Other source-to-source translation problems that may be solvable with C-to-C include automatic or programmer-directed source-level optimization, program analysis, source code metrics, error checking (e.g., `lint` tools), and instrumentation for profiling or debugging.

6.2 Getting C-to-C

The source code of C-to-C is freely available from `ftp://theory.lcs.mit.edu/pub/c2c`. Building C-to-C requires an ANSI C compiler and the free GNU tools `flex` and `bison`.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings 7th SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[3] American National Standards Institute, Inc., NY. *American National Standards for Information Systems, Programming Language C ANSI X3.159-1989*. 1990.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Phil Lisiecki, Rob Miller, Keith H. Randall, Andy Shaw, and Yuli Zhou. *Cilk 2.0 Reference Manual*. MIT LCS, 545 Technology Square, Cambridge, MA 02139, May 1995. Available via `ftp://theory.lcs.mit.edu/pub/cilk/manual-2.0.ps.Z`.

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995. To appear.

[6] François Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwais, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In *Proceedings of OONSKI '94*, 1994. Also available as ftp://ftp.extreme.indiana.edu/pub/sage/oonski94.ps.gz.

[7] François Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[8] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object oriented programming notation. Technical Report CS-TR-92-01, California Institute of Technology, Pasadena, CA, 1992.

[9] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93, the ACM/IEEE Conference*, Portland, OR, November 1993.

[10] Chris Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.

[11] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23:158–171, 1976.

[12] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–318, 1977.

[13] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, 2nd Edition*. Prentice Hall, 1988.

[14] Gary Kildall. A unified approach to global program optimization. In *Proceedings 1st ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[15] J. R. Rose and G. L. Steele Jr. C*: An extended language for data parallel programming. In *Proceedings 2nd International Conference on Supercomputing*, volume 2, pages 2–16, San Francisco, CA, May 1987.

[16] Richard Stallman et al. GNU C compiler (GCC) version 2. Source code and documentation available from `prep.ai.mit.edu` via anonymous ftp, in directory `/pub/gnu.`, 1994.