Tree-Based Code Optimization

A Thesis Proposal

Carl McConnell
University of Illinois at Urbana-Champaign
E-mail: mcconnel@cs.uiuc.edu

May 28, 1992

Abstract

Nearly all algorithms for code optimization use a control flow graph. In this thesis, I will show that with very minor restrictions on program structure, an abstract syntax tree can be used instead, leading to algorithms that are often much simpler than their graph-based counterparts. The conclusion is that abstract syntax trees, not control flow graphs, should be the fundamental data structure in code optimization.

1 Introduction

Most optimizing compilers consist of a front-end that does syntactic and semantic analysis, and a back-end that does optimization and machine code generation [ASU86]. The main data structure in the front-end is an abstract syntax tree (AST), while in the back-end it is a control flow graph (CFG), which consists of nodes representing computations and edges representing control flow. Thus, code optimization operates on a data structure—the CFG—in which the program has essentially been "flattened" into a tangle of `GOTO`s (edges). Unlike an AST, a CFG cannot directly express the hierarchical organization of a program, nor the high-level control structures like `WHILE-DO` and `IF-THEN` that impose this organization.

However, it is these high-level control structures that are of interest in most optimizations! For example, loops are the focus of optimizations such as code motion and strength reduction. However, since they are not explicit in CFGs, they must be uncovered using an algorithm requiring *dominator* information [LT79] [Har85]: a node D *dominates* a node N if every path from the initial node of the CFG to N goes through D. (Dominator information may be represented as a tree called the *dominator tree.*) Even register allocation can benefit from knowledge of a program's hierarchical control structure [CK91]. CFGs oblige us to recover this structure, which is usually explicit at the source level. If we could somehow avoid this salvage effort, we would end up with simpler optimization algorithms, and hence simpler compilers.

In this thesis I will show how to use an AST instead of a CFG for code optimization. We will see that an AST-based code optimization algorithm is often simpler than the corresponding CFG-based algorithm. This is because the structure of an AST makes CFG concepts such as topological ordering and dominators unnecessary. Since simplicity is in the eye of the beholder, we will measure it according to the number of auxiliary data structures an algorithm requires: for example, many CFG-based algorithms use a dominator tree, while AST-based algorithms do not. Finally, to show that these ideas are useful in doing even low-level optimization, I will describe their implementation in the RTL System code optimizer and generator [JM91].

Powell [Pow84] states that "A survey of the literature on optimization uncovers many techniques that are hard to understand, harder to implement, more general than necessary, and of marginal value." I agree with this assessment, and believe that the CFG is the main cause of the first two afflictions.

2 Kicking the Control Flow Graph Habit

Why are CFGs treated as a *sine qua non* of code optimization? One reason is probably historical in nature. According to [Knu73], Goldstine and von Neumann began using flow charts to describe programs in 1946. This apparently set a precedent that subsequent workers followed with little question. Another more important reason for the prevalence of CFGs is that they describe control flow in a way that is easy to express with the `GOTO`-like branching instructions computers have. Yet another reason is the infamous

GOTO statement itself, which is a feature of many popular programming languages. Although ASTs can and do express GOTOs, their tree structure does not reflect the potential flow of control: by definition, each sub-tree in an AST has a single entry point, but GOTOs can cause control to jump into the middle of a sub-tree. (Shortly, though, we will take advantage of the fact that this almost never happens in practice.) On the other hand, CFGs nicely describe the arbitrary control flow that GOTOs can induce, and so all other control structures are expressed in terms of GOTO as well. Thus, the GOTO tail wags the control structure dog [Ros77]: high-level control structures are sacrificed for the sake of GOTOs. However, as Powell says [Pow84], "The value of constructing flow graphs for languages or programs without GOTO statements (as well as the payoff from considering all the perverse possibilities in languages that do have them) is minimal."

(The preceding remarks concerning the hegemony of CFGs are only true in the procedural language community. The Lisp/Scheme community uses ASTs in their compilers [Ste78] [Kra88]. For the most part, however, code optimization to them means applying closure analysis techniques to reduce a program to a procedural form, at which point their compilers generate code. They do not discuss how to perform traditional optimizations using ASTs. Thus, their point of arrival is our point of departure.)

One way to avoid constructing CFGs, and thus to avoid sacrificing control structures, is to use the AST of the front-end in the back-end as well. However, there are two problems with this idea. One is that this AST is language-dependent, while the back-end should be language-independent. The other problem is that the control structures described in this AST are generally too high-level to expose opportunities for low-level optimization: a Pascal FOR-DO statement, for example, does not correspond to an instruction on most machines.

To solve these problems, I introduce an intermediate language that for convenience I will call **CORTL**, for "COntrol structures even at the Register Transfer Level." **CORTL** can easily express the control structures used in languages such as C or Fortran without forfeiting the hierarchical organization they impose on a program, and a **CORTL** AST is as easy to generate from the source language AST as a CFG is. At the same time, the **CORTL** control structures correspond closely to the branching instructions that machines have. This allows **CORTL** ASTs to be used for low-level optimization in the back-end. Figure 1 shows the abstract syntax of **CORTL**. The con-

$$
\begin{array}{rcl}
S & \rightarrow & \texttt{region}\ I\ S \\
 & | & \texttt{depart}\ I\ \texttt{after}\ S \\
 & | & \texttt{repeat}\ I\ \texttt{after}\ S \\
 & | & \texttt{if}\ E\ \texttt{then}\ S \\
 & | & S\ S \\
 & | & I\ \texttt{:=}\ E \\
 & | & \texttt{nothing} \\
 & & \\
I & \rightarrow & \text{identifier} \\
E & \rightarrow & \text{expression}
\end{array}
$$

Figure 1: Abstract syntax of **CORTL**

ditional, assignment, and sequencing statements have the usual meaning. The `region` and `depart` constructs are similar to the label and `leave` constructs of Bliss [WJW*75]: `region` I S declares a region of code named I consisting of S, and `depart` I `after` S executes S and then jumps to the statement after region I. On the other hand, `repeat` I `after` S executes S and then jumps *to* region I, thus providing a means of doing iteration; there is no looping construct as such. The keyword `nothing` denotes the empty statement, and is only useful with `depart` or `repeat`, as in `depart` I `after nothing`. Since doing `nothing` is so common, for brevity we will often omit the `after` part, as in `depart` I. It might seem simpler to omit `depart` I `after` S from the abstract syntax and just use S followed by `depart` I instead. However, in order to simplify our algorithms, we will require that `depart`s and `repeat`s do not occur in the midst of statement lists.

CORTL embodies ideas that arose during the structured programming debate, which [Knu74] discusses. To summarize, during the late 1960s and early 1970s the programming community gradually realized that using control structures such as `IF-THEN` and `WHILE-DO` instead of the ubiquitous `GOTO` improved program clarity [Dij68]. At the same time, it became apparent that not all uses of `GOTO` were harmful. For example, using a `GOTO` to perform a multi-level exit from nested `WHILE-DO` loops was much clearer than the `GOTO`-less alternative; it was generally `GOTO`s *into* control structures, not *out* of them, that were harmful. In [PKT73], Peterson *et al.* defined a *well-*

formed program to be one in which all control structures have a single entry point, and showed that such a program need only use `IF-THEN`, `DO-FOREVER`, and multi-level loop `EXIT` statements as control structures. The single-entry property of well-formed programs makes analysis of them simpler.[1] It also makes an AST representation more attractive, since the resulting control flow is tree-structured [Weg73]: control enters a sub-tree only at its root. For these reasons, **CORTL** uses what amounts to the same control structures.

By way of proof that **CORTL** can express the control structures of any procedural language, Figure 2 shows the **CORTL** equivalents of representatives. The templates in the left column span the range of non-`GOTO` constructs used in procedural languages such as Fortran and Pascal. (Although not shown, well-behaved `GOTO`s such as the `break` and `continue` statements of the C programming language are obviously expressible as `depart` and `repeat` statements.) These examples show that translating a control structure into **CORTL** only increases its size by a small additive factor. Thus, translating a program having only well-behaved `GOTO`s into **CORTL** will not cause a polynomial or exponential "blow-up" that would make **CORTL** unsuitable for representing programs.

3 Dealing with `GOTO`s

Although `GOTO`s occur infrequently, they do crop up, and must be dealt with somehow. `GOTO`s come in two varieties: structured (those that do not jump into the middle of control structures) and unstructured (those that do). Translating structured `GOTO`s into `depart` and `repeat` statements poses no problem. On the other hand, unstructured `GOTO`s clash with the premise underlying **CORTL**: that control structures are entered at the top. Therefore, they must be turned into structured `GOTO`s. This can be done by duplicating code ("node splitting"), which can cause an exponential increase in program size, but is unlikely to do so in practice. In the thesis I will give an algorithm for translating programs with `GOTO`s of both varieties into **CORTL**.

How often do unstructured `GOTO`s occur? The answer is: rarely. Kennedy analyzed 500 Fortran routines to find out what percentage were what he called *semi-structured* [KZ77], which is equivalent to being well-formed. He found that 82% fell into this category. Since the time of that study (1977),

[1] Note that the well-formed programs are a proper subset of the reducible ones [Hec77].

Control structure	**CORTL** equivalent

`if` E `then`
  S_1
`else`
  S_2
	`region R` $\{$
	  `if` E `then`
	    `depart R after` S_1`;`
	  S_2`;`
	$\}$

`repeat`
  S
`until` E
	`region R` $\{$
	  S`;`
	  `if` E `then`
	    `depart R;`
	  `repeat R;`
	$\}$

`while` E `do`
  S
	`region R`
	  `if` E `then`
	    `repeat R after` S`;`

`case` E `in`
  E_1`:` S_1
  ...
  E_n`:` S_n
  `default:` $S_{default}$
	`region R` $\{$
	  `if` E `=` E_1 `then`
	    `depart R after` S_1`;`
	  ...
	  `if` E `=` E_n `then`
	    `depart R after` S_n`;`
	  $S_{default}$`;`
	$\}$

Figure 2: **CORTL** equivalents of representative control structures

this percentage has no doubt increased thanks to the widespread adoption of structured programming techniques. Furthermore, unstructured `GOTO`s are almost never seen in programs written in other languages, such as C or Pascal; Fortran programs are no doubt the worst offender in this regard. Thus, to belabor an earlier metaphor, the `GOTO` tail is a very small part of the control structure dog, and we can treat it accordingly.

In what follows, we will use Kennedy's terminology and refer to well-formed programs as semi-structured. (Structured programs, of course, are those expressible using only `WHILE-DO`, `IF-THEN`, and sequencing statements.)

4 Preliminaries

Before proceeding further, it will be helpful to establish some ground rules:

- We will often deliberately confuse statements with the AST nodes that represent them.
- A region that is the target of one or more `repeat` statements is a *loop region* or *loop*, and otherwise is a *non-loop region*.
- Without loss of generality, we will assume that region names are unique.
- Statement lists will be represented using Lisp-style pairs. A `nothing` statement called `Nothing` will denote the end-of-list marker, exactly as `nil` does in Lisp-style lists.
- Except where needed in list manipulation, the existence of `nothing` statements will usually be ignored in algorithms and elsewhere, since they do ... nothing.
- We will assume a variable called `Root` contains the root of the AST.
- We will let P denote the size of the AST/program. Thus, a linear algorithm is $O(P)$.
- The algorithms assume AST nodes have attributes accessed using ".", as in $X.child$. Figure 3 shows the items in the abstract syntax, the corresponding AST node types, and their chief attributes; other attributes will be introduced as necessary.

Item	AST node type	Attributes
`region` I S	`region`	*child*: AST node for S
`depart` I `after` S	`depart`	*child*: AST node for S *region*: AST node for `region` I
`repeat` I `after` S	`repeat`	*child*: AST node for S *region*: AST node for `region` I
`if` E `then` S	`if-then`	*condition*: expression for E *child*: AST node for S
S S	`list`	*first*: AST node for first statement in list *rest*: AST node for rest of list
I `:=` E	`assignment`	*destination*: identifier for I *source*: expression for E

Figure 3: **CORTL** AST nodes

- In the algorithms, `new XNode(c1, c2, ...)` creates a new AST node of type `X` with "chief" attribute values `c1`, `c2`, ... For example, `new ListNode(S, Nothing)` creates a new statement list containing only one statement, `S`, and is analogous to `(cons S nil)` in Lisp.

The obvious way to implement **CORTL** ASTs is to use records to represent nodes, and their components to represent attributes. However, it can be advantageous to instead represent an attribute A by a mapping from an arbitrary AST node X to $X.A$. The foremost example of this is the *region* attribute: if represented as a record component, it would turn the tree into a graph, making it more difficult to manipulate. In practice, it would likely be represented by a record component giving only the name of the region, which could then be mapped to the corresponding AST node using a dictionary.

To simplify the algorithms, we will assume that statement lists do not contain `depart` or `repeat` statements. In other words, occurrences of

S_1
...
S_n
`depart` I

must be converted to

```
depart I after {
  S_1
  ...
  S_n
}
```

before applying any algorithm, and likewise with `repeat` statements. Furthermore, we will assume that `region`s are explicitly `depart`ed. In other words, occurrences of

```
region I S
```

must be converted to

```
region I
  depart I after S
```

before optimization, unless S is already `depart` I or `repeat` I.

5 Data Dependences and SSA Form

Optimization requires information about the data dependences in a program: which assignments define the values used by a given assignment? This question is easy to answer if each variable is assigned a value exactly once, and this is what static single assignment (SSA) form, introduced by Cytron *et al.* [CFR*91], guarantees. "Single assignment" languages are functional languages in which each variable is assigned a value exactly once even at run-time, not merely in the program text. In contrast, any program can be put into SSA form; it is merely an alternative representation. However, Cytron *et al.* based their treatment of SSA form on CFGs, so a new SSA conversion algorithm based on ASTs is necessary. We will see that the AST-based algorithm is much simpler than the original.

It is easy to rename variables in straight-line code so that there is only one assignment to (i.e., definition of) each variable. However, this is not possible in the presence of conditionals or loops, since a given point in the program may be reached by conflicting definitions. In SSA form, a program point reached by a set of conflicting definitions has a *ϕ-assignment* that merges them. A ϕ-assignment has the form $U := \phi(V, W, \ldots)$, where $U, V, W, \ldots$

are variables, and the number of operands $V, W, \ldots$ corresponds to the number of control paths to that ϕ-assignment. If control reaches a ϕ-assignment through the j-th path, then U is assigned the value of the j-th operand. Each execution of a ϕ-assignment uses only one of its operands, but which one depends on how control reaches that ϕ-assignment. In **CORTL**, ϕ-assignments can only occur after `if-then` statements, after `region` statements, and just inside loops, since these are the only situations in which multiple control paths merge. We will refer to these kinds of ϕ-assignments respectively as *if-then*, *region*, and *loop* ϕ-assignments, and call the statements possessing them *ϕ-bearing*. Figure 4 shows a program before and after SSA conversion.

The SSA conversion algorithm requires several new attributes. For an AST node S, *S.variablesDefined* is the set of variables having definitions in the sub-tree rooted at S that would reach a statement immediately following S. For example, consider

```
A region R
B   depart R after {
C     if E1 then
        v := 1
D     if E2 then
E       depart R after
          w := 1
    }
```

For any `depart` or `repeat` statement X, *X.variablesDefined* is always empty, since the definitions under X can never reach the succeeding statement. Therefore, *B.variablesDefined* = *D.variablesDefined* = *E.variablesDefined* = $\emptyset$. However, *A.variablesDefined* = {`v, w`} even though *B.variablesDefined* is empty. Algorithm **VariablesDefined** in Figure 5 computes this attribute. Its execution time is $O(P)$.

The other new attributes are just numberings. We assume the `depart` (resp. `repeat`) statements that reference a region R are consecutively numbered from 1 to *R.departCount* (resp. *R.repeatCount*), and that the number of a `depart` (resp. `repeat`) statement X is *X.departNumber* (resp. *X.repeatNumber*). The order in which they are numbered is unimportant. Note that *repeatCount* can be used to distinguish loop regions from non-loop regions: a region R is a non-loop region iff *R.repeatCount* is zero. Algorithm **DepRepCount** in Figure 6 computes the values of these attributes. Its execution time is $O(P)$.

```
i := 1
j := 1
region R {
  if E1 then
    depart R after
      j := j + 1
  i := j
  region S {
    if E2 then
      depart S after
        i := i + 2
    if E3 then
      repeat R after
        i := i + 3
    if E4 then
      depart R after
        j := j + 4
  }
  if E5 then
    j := j + 5
  repeat R
}
```

```
i.1 := 1
j.1 := 1
region R {
  i.2 := φ(i.1, i.5, i.7)
  j.2 := φ(j.1, j.2, j.6)
  if E1 then
    depart R after
      j.3 := j.2 + 1
  i.3 := j.2
  region S {
    if E2 then
      depart S after
        i.4 := i.3 + 2
    if E3 then
      repeat R after
        i.5 := i.3 + 3
    if E4 then
      depart R after
        j.4 := j.2 + 4
    i.6 := i.3 + 1
  }
  i.7 := φ(i.4, i.6)
  if E5 then
    j.5 := j.2 + 5
  j.6 := φ(j.2, j.5)
  repeat R
}
i.8 := φ(i.2, i.3)
j.7 := φ(j.3, j.4)
```

Figure 4: **CORTL** program before and after SSA conversion

```
VariablesDefined(Root);

VariablesDefined(thisNode)
  thisNode.variablesDefined ← new Set;
  for each child c of thisNode do
    VariablesDefined(c);
  if thisNode is a depart or repeat node then
    Add thisNode.child.variablesDefined to thisNode.region.variablesDefined;
  else if thisNode is an assignment node then
    Add thisNode.destination to thisNode.variablesDefined;
  else
    for each child c of thisNode do
      Add contents of c.variablesDefined to thisNode.variablesDefined;
end procedure
```

Figure 5: Algorithm **VariablesDefined**

The algorithm for SSA conversion consists of two parts: inserting trivial ϕ-assignments of the form `v :=` ϕ`(v, ..., v)`, and renaming variables. For any ϕ-bearing statement S, let *S.phiAssignments* represent either the if-then or region ϕ-assignments, as appropriate; and let *S.loopPhiAssignments* represent the loop ϕ-assignments, if appropriate. If-then ϕ-assignments only have two arguments; the region ϕ-assignments for a region S have *S.departCount* arguments; and the loop ϕ-assignments for a loop region S have *S.repeatCount* + 1 arguments. Determining which variables require ϕ-assignments is easy: for a ϕ-bearing statement S, a ϕ-assignment is required for each variable in *S.variablesDefined*. Algorithm **InsertPhi** in Figure 7 inserts trivial phi assignments. Its execution time is $O(VP)$, where V is the number of variables in the program. Thus, the algorithm could take (and could increase program size by) $O(P^2)$. However, V will be much less than P for all but the smallest programs. Thus, Algorithm **InsertPhi** will normally both run in (and increase program size by) only $O(P)$. Cytron *et al.* present empirical evidence supporting this claim in [CFR*91]. In what follows, P denotes the size of the program after ϕ-assignments have been added.

Renaming variables simply involves traversing the AST while maintaining a renaming dictionary and a counter for each variable. The only complication

```
DepRepCount(Root);

DepRepCount(thisNode)
  if thisNode is a region node then
    r.departCount ← r.repeatCount ← 0;
  for each child c of thisNode do
    DepRepCount(c);
  case thisNode is
    depart node:
      n ← thisNode.region.departCount + 1;
      thisNode.region.departCount ← n;
      thisNode.departNumber ← n;

    repeat node:
      n ← thisNode.region.repeatCount + 1;
      thisNode.region.repeatCount ← n;
      thisNode.repeatNumber ← n;
  end case
end procedure
```

Figure 6: Algorithm **DepRepCount**

```
InsertPhiAssignments(Root);

InsertPhiAssignments(thisNode)
  for each child c of thisNode do
    InsertPhiAssignments(c);
  case thisNode is
    if-then node:
      thisNode.phiAssignments ← MakeTrivials(thisNode.variablesDefined, 2);

    region node:
      thisNode.phiAssignments ← MakeTrivials(thisNode.variablesDefined,
        thisNode.departCount);
      if thisNode is a loop region node then
        thisNode.loopPhiAssignments ← MakeTrivials(thisNode.variablesDefined,
          thisNode.repeatCount);
  end case
end procedure

MakeTrivials(varsDefined, argCount)
  result ← Nothing;
  for each variable v in varsDefined do
    result ← new ListNode(trivial φ-assignment for v
      having argCount arguments, result);
  return result;
end procedure
```

Figure 7: Algorithm **InsertPhi**

involves `depart` and `repeat` statements: the effects of renaming in a sub-tree rooted at one of these should not propagate out of the sub-tree, so a partial copy of the renaming dictionary is made. Algorithm **SSA** in Figure 8 renames variables so that the program is in SSA form. Its execution time is $O(P)$.

The SSA conversion algorithm given here demonstrates how using an AST can simplify an algorithm. Although the CFG-based algorithm given by Cytron *et al.* has the advantage of working for arbitrary CFGs, I believe it is much harder to comprehend than the one given here. Furthermore, it requires two auxiliary data structures—the dominator tree and a related structure called the dominance frontier mapping—while the AST-based algorithm does not.

We will use the following terminology when referring to data dependences: if an assignment A defines values that an assignment or if-then statement S uses, then S is a *flow dependent* of A, and A is a *flow supporter* of S. A.*flowDependents* gives the flow dependents of A, and S.*flowSupporters* gives the flow supporters of S.

6 Control Dependence

The notion of *control dependence* [FOW87] plays an important role in many optimizations, such as moving computations out of loops. We will say a statement S is control dependent on a statement C if

- C is an `if` E `then` statement; and
- C lexically precedes S; and
- when the value of E is b, S will definitely execute, but when it is $\neg b$, S may or may not execute; the control dependence is assigned the value b.

For example, in

```
region R {
  if E then
    depart R after S1
  S2
```

```
DefinitionCount ← new Dictionary;
SSA(Root, new Dictionary);

SSA(thisNode, renames)
  case thisNode is
    region node:
      if thisNode is a loop region node then
        for each assignment a in thisNode.loopPhiAssignments do begin
          RenamePhiOperand(1, a, renames);
          RenameLHS(a, renames);
      end;
      SSA(thisNode.child, renames);
      for each assignment a in thisNode.phiAssignments do
        RenameLHS(a, renames);

    if-then node:
      for each assignment a in thisNode.phiAssignments do
        RenamePhiOperand(1, a, renames);
      SSA(thisNode.child, renames);
      for each assignment a in thisNode.phiAssignments do begin
        RenamePhiOperand(2, a, renames);
        RenameLHS(a, renames);
      end;

    depart node:
      newRenames ← new Dictionary;
      for each assignment a in thisNode.region.phiAssignments do
        newRenames[a.destination] ← renames[a.destination];
      SSA(thisNode.child, newRenames);
      for each assignment a in thisNode.region.phiAssignments do
        RenamePhiOperand(thisNode.departNumber, a, newRenames);

continued ...
```

Figure 8: Algorithm **SSA**

```
    repeat node:
      SSA(thisNode.child, renames);
      for each assignment a in thisNode.region.loopPhiAssignments do
        RenamePhiOperand(thisNode.repeatNumber, a, renames);

    list node:
      SSA(thisNode.first, renames);
      SSA(thisNode.rest, renames);

    assignment node:
      DefinitionCount[v] ← 0;
      for each variable v used in thisNode.source do
        Replace v with v.renames[v];
      RenameLHS(thisNode, renames);
  end case
end procedure

RenamePhiOperand(n, a, renames)
  Replace the n-th operand v of the ϕ-function in a with v.renames[v];
end procedure

RenameLHS(a, renames)
  v ← a.destination;
  i ← DefinitionCount[v] + 1;
  DefinitionCount[v] ← i;
  a.destination ← v.i;
  renames[v] ← i;
end procedure
```

Figure 8: (continued)

}
S_3

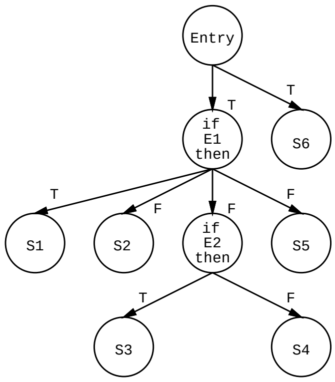
S_1 is control dependent on `if` E `then` with value *true*, S_2 is control dependent on it with value *false*, and S_3 is not control dependent on it at all. The definition of control dependence in CFGs is given in terms of post-dominators to make computing control dependence easier [FOW87], but the intuition behind it is the same; the lack of structure in CFGs precludes basing an algorithm on the definition given here. The CFG definition includes the backward control dependences caused by loops. We exclude them, since backward data dependences make them redundant.

Control dependence edges together with the AST nodes they connect form a DAG called the *control dependence graph*[2] (CDG) whose root is a node labelled `Entry`; all edges leaving `Entry` are labelled *true*. We will only consider `if-then` and assignment statements when computing control dependences, since any optimizations involving these must correctly maintain relationships among other kinds of statements anyway. Figure 9 shows some **CORTL** programs and their CDGs. When edges representing data dependences are added to the CDG, the result is essentially the *program representation graph* (PRG) of [YHR89], which in turn is similar to the *program dependence graph* of [FOW87].

In structured programs, computing control dependences is trivial: as noted in [HPR89], a statement is only control dependent on its nearest `if-then` ancestor in the AST. In **CORTL** programs, though, this is not always true: as Figure 9 shows, there is often no simple relationship between an `if-then` statement and the statements that are control dependent on it. Algorithm **ControlDep** in Figure 11 computes control dependences for **CORTL** programs. It uses the *regionsDeparted* and *regionsRepeated* attributes computed by Algorithm **DepRepRegions** in Figure 10: *S.regionsDeparted* (resp. *S.regionsRepeated*) gives the regions surrounding S that are referred to by `depart` (resp. `repeat`) statements under S. If either *S.regionsDeparted* or *S.regionsRepeated* is not empty, then the statement after S might not execute. We will refer to such a statement S as "jumpy".

[2]Since we are ignoring backward control dependences, this is technically the *forward* control dependence graph [CHH89].

```
region A {
  region B {
    if E1 then
      depart A after S1
    S2
    if E2 then
      depart B after S3
    S4
  }
  S5
}
S6
```




```
region A {
  region B {
    if E1 then
      depart B after S1
    S2
    if E2 then
      depart A after S3
    S4
  }
  S5
}
S6
```

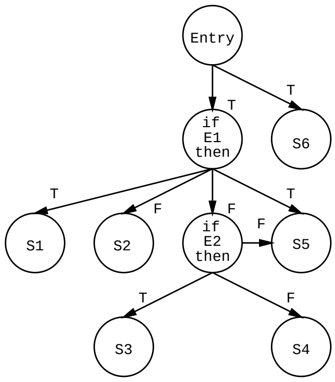


Figure 9: Some **CORTL** programs and their control dependence graphs

Algorithm **ControlDep** traverses the AST while representing the immediate CDG predecessors of the current AST node—the *conditions*—as a mapping from `if-then` (or `Entry`) nodes to boolean values; that is, if the algorithm is currently considering S, then for each key k in *conditions*, there will be an edge from k to S labelled *conditions[k]*. In the absence of either statement lists or `depart` statements, every statement would only be control dependent on an ancestor, as with structured programs; in other words, the conditions would only be "inherited" (passed downward). However, a jumpy statement may alter the control dependences for subsequent statements, as Figure 9 demonstrates. Therefore, `ControlDep()` must return the conditions appropriate for the next statement in the statement list, if any. These "synthesized" conditions differ from the inherited conditions in only two situations:

- By definition, a statement following a jumpy if-then statement C is control dependent on C with value *false*. Hence, if an `if-then` node S is jumpy, the algorithm augments the conditions produced by *S.child*. These conditions are empty in the base case where *S.child* is a `depart` or `repeat` node, as is appropriate.

- The control dependences for a statement following a region statement R depend on the conditions under which R is `depart`ed, represented by *R.departingConditions*. Whenever a depart statement referencing R is encountered, the current conditions are recorded in *R.departingConditions* using the `Merge()` function. This function checks to see if R is being `depart`ed under inconsistent conditions; that is, under both *true* and *false* values of the predicate of the same `if-then` node C. If so, there is not really a control dependence involving C, in which case there may be one involving C's CDG predecessors. To handle this possibility, `Merge()` calls itself. The recursion stops when a valid control dependence is found, or when `Entry` is reached.

The execution time of Algorithm **ControlDep** is $O(P^2)$. However, for the **CORTL** equivalent of structured programs it is $O(P)$. Thus, the expected execution time is linear.

We will use terminology similar to that for data dependences when referring to control dependences: if a statement S is control dependent on an if-then statement C, then S is a *control dependent* of C, and C is a *control*

```
DepRepRegions(Root);

DepRepRegions(thisNode)
  thisNode.regionsDeparted ← new Set;
  thisNode.regionsRepeated ← new Set;
  for each child c of thisNode do begin
    DepRepRegions(c);
    for each r in c.regionsDeparted do
      Add r to thisNode.regionsDeparted;
    for each r in c.regionsRepeated do
      Add r to thisNode.regionsRepeated;
  end
  if thisNode is a region node then begin
    if thisNode is in thisNode.regionsDeparted then
      Remove thisNode from thisNode.regionsDeparted;
    if thisNode is in thisNode.regionsRepeated then
      Remove thisNode from thisNode.regionsRepeated;
  end
end procedure
```

Figure 10: Algorithm **DepRepRegions**

supporter of S. $C.controlDependents$ gives the control dependents of C, and $S.controlSupporters$ gives the control supporters of S.

7 Common Subexpression Elimination

Common subexpression elimination (CSE) replaces one expression with another (typically a variable) that has the same value, but that is cheaper to compute. Unlike traditional CSE algorithms, Algorithm **CSE** in Figure 12 can do this even in the presence of control structures.

At the heart of Algorithm **CSE** is a partitioning algorithm (not shown) borrowed from [AWZ88] [AHU74] [YHR89], and that is based on Hopcroft's algorithm [Hop71] for minimizing finite state machines. The partitioning algorithm takes an initial partition consisting of equivalence sets (sets of graph

```
Compute regionsDeparted and regionsRepeated attributes;
conditions ← new Dictionary;
conditions[Entry] ← true;
ControlDep(Root, conditions);

ControlDep(thisNode, conditionsAbove)
  case thisNode is
    region node:
      S.departingConditions ← new Dictionary;
      if thisNode is a loop region then
        AddManyEdges(conditionsAbove, thisNode.loopPhiAssignments);
      ControlDep(thisNode.child, conditionsAbove);
      conditionsAfter ← thisNode.departingConditions;
      AddManyEdges(conditionsAfter, thisNode.phiAssignments);

    if-then node:
      AddEdges(conditionsAbove, thisNode);
      conditionsBelow ← new Dictionary;
      conditionsBelow[thisNode] ← true;
      conditionsAfter ← ControlDep(thisNode.child, conditionsBelow);
      if (thisNode.regionsDeparted is not empty) or
      (thisNode.regionsRepeated is not empty) then
        conditionsAfter[thisNode] ← false;
      AddEdges(conditionsAbove, thisNode.phiAssignments);

    list node:
      conditionsAfter ← ControlDep(thisNode.first, conditionsAbove;
      conditionsAfter ← ControlDep(thisNode.rest, conditionsAfter);
```

continued …

Figure 11: Algorithm **ControlDep**

```
    depart node:
      for each key k in conditionsAbove do
        Merge(k, conditionsAbove[k], thisNode.regionNode.departingConditions);
      ControlDep(thisNode.child, conditionsAbove);
      conditionsAfter ← new Dictionary;

    repeat node:
      ControlDep(thisNode.child, conditionsAbove);
      conditionsAfter ← new Dictionary;

    assignment node:
      AddEdges(conditionsAbove, thisNode);
      conditionsAfter ← conditionsAbove;
  end case
  return conditionsAfter;
end function

Merge(ifNode, value, conditions)
  if conditions has no entry for ifNode then
    conditions[ifNode] ← value;
  else for each control dependence edge e entering ifNode do
    Merge(source of e, label of e, conditions);
end procedure

AddEdges(conditions, node)
  for each key k in conditions do
    Add an edge labelled conditions[k] from k to node;
end procedure

AddManyEdges(conditions, nodes)
  for each AST node tn in nodes do
    AddEdge(conditions, tn);
end procedure
```

Figure 11: (continued)

```
As described in the text, create an initial partition P consisting of
sets of if-then and assignment AST nodes;

Apply the partitioning algorithm to P, refining it into
sets of nodes with equivalent control and data dependences;

for each set s in P do
  if s consists of assignments then begin
    a ← the member of s that is lexically first in the program;
    for each assignment b in s do
      if b ≠ a then
        Replace all uses of b.destination by a.destination;
  end;

Eliminate any resulting dead code;
```

Figure 12: Algorithm **CSE**

nodes assumed to be equivalent), and refines it until every equivalence set consists only of nodes that were in the same equivalence set initially, and whose predecessors are in the same equivalence set. Thus, by applying the partitioning algorithm to the PRG, we end up with sets of essentially interchangeable nodes. Hence, given an assignment A that lexically precedes the other assignments in its equivalence set, we can replace a use of the destinations of those other assignments with a use of the destination of A. When followed by dead code elimination, this results in the elimination of redundant control structures. The execution time of Algorithm **CSE** is dominated by that of the partitioning algorithm, which is $C \log C + D \log D$, where C and D are respectively the number of control and data dependence edges between the AST nodes. Figure 13 shows an example of Algorithm **CSE** in action.

It remains to describe how the initial partition is constructed from the AST nodes:

- There is an equivalence set containing all `if-then`s.
- There are equivalence sets containing all non-ϕ assignments having the

```
region R {
  if E then
    depart R after
      i.1 := 1
  i.2 := 2
}
i.3 := φ(i.1, i.2)
region S {
  if E then
    depart S after
      j.1 := 1
  j.2 := 2
}
j.3 := φ(j.1, j.2)
k.1 := i.3 + j.3
```

```
region R {
  if E then
    depart R after
      i.1 := 1
  i.2 := 2
}
i.3 := φ(i.1, i.2)
k.1 := i.1 + i.1
```

Figure 13: A program before and after Algorithm **CSE** executes

same operator on the right-hand side.

- There is an equivalence set for all if-then ϕ-assignments having the same number of arguments.

- There is an equivalence set for all region ϕ-assignments having the same number of arguments.

- There is an equivalence set for all loop ϕ-assignments having the same number of arguments.

Algorithm **CSE** is essentially the same as the algorithm Yang *et al.* give in [YHR89] for detecting program components with equivalent behaviors. However, that algorithm is limited to a structured programming language having only `IF-THEN-ELSE` and `WHILE-DO` control structures. It also unnecessarily uses a CFG to determine where to insert ϕ-assignments. Alpern *et al.* [AWZ88] give a global CFG/SSA-based CSE algorithm that can accommodate high-level constructs, although it does so in an *ad hoc* way by labelling ϕ-assignments in the CFG according to the construct that produced them, as well as by adding additional ϕ arguments to indicate control dependence. They are rather vague as to how these things are to be accomplished.

8 Code Motion out of Loops

An assignment or if-then statement is *invariant* with respect to a loop L if all of its flow and control supporters are either invariant with respect to L or occur outside L. A standard optimization called *code motion* (CM) moves invariant assignments out of loops, thus reducing program execution time. Traditional CM algorithms are stymied by control structures: for example, they cannot move invariant assignments embedded in an invariant `IF-THEN`. Shortly we will give an AST-based CM algorithm that overcomes this limitation. First, however, we describe a loop transformation that enhances the usefulness of the algorithm.

Under the definition of invariance given above, loops with a loop-departing statement[3] near the top of the loop contain little if any invariant code, since

[3]A statement S in a loop region L is *loop-departing* if $L \in S.regionsDeparted$, and analogously for *loop-repeating*.

most of the loop body is control dependent on the loop-departing statement. The most common kind of loop, the `WHILE-DO` loop, suffers from this drawback. Therefore, prior to SSA conversion it is advantageous to perform a transformation we will call *partial loop peeling* to "expose" loop bodies to CM. In (full) loop peeling, the entire loop body is copied. Partial loop peeling consists of placing a copy of the statements up to and including "the" loop-departing statement—selecting one of several, if necessary—just before the loop, and then rotating the body of the loop so that these statements come at the end of the loop, thus avoiding redundant execution of them on the first iteration. Copying does not proceed past a loop-repeating statement, since any invariant code appearing after such a statement cannot be moved out of the loop anyway.

Algorithm **PartialPeel** in Figure 15 performs partial loop peeling. It assumes the loop body is a statement list. To minimize code expansion, it only copies everything up to and including the first loop-departing statement, if any. Figure 14 shows its effect. The execution time (and the effect on program size) of Algorithm **PartialPeel** is potentially exponential, since it can end up duplicating the body of every loop. However, this behavior seems extremely unlikely; in practice, the execution time should be $O(P)$, and the effect on program size quite small.

Algorithm **CodeMotion** in Figure 16 performs CM on programs in SSA form. It assumes the loop body is a statement list, and it stops when it reaches a loop-departing or loop-repeating statement, since any invariant code appearing after such a statement cannot be moved out of the loop. Otherwise, the algorithm simply proceeds in a bottom-up fashion, marking the invariant code in each loop, and moving it and a copy of the control structures surrounding it to just in front of the loop. In dealing with subloops, it makes use of an *inputVariables* attribute, which for a loop L gives the set of variables used inside L but defined outside L. If the definitions of the variables in $L.inputVariables$ are invariant, then the values that L computes are also invariant, and the entire loop may be treated as an invariant (albeit complex) assignment statement. The *inputVariables* attribute can be computed in linear time as the algorithm progresses, although the details are omitted for the sake of brevity. Also omitted are the straightforward but tedious details of updating control and data dependences. (For example, what happens if only some of the flow supporters of a ϕ-assignment are invariant?) Since the algorithm may have created empty control structures, as a last step

```
region R
  repeat R after {
    i := i + 1
    if i > n then
      depart R
    region S {
      if n = 1 then
        depart S after
          a := 7
      a := 8
      i := i + 1
    }
    if i = 0 then
      depart R
    c := 3
    m := a + c
  }
```

```
region Q {
  i := i + 1
  if i > n then
    depart Q
  region R
    repeat R after {
      region S {
        if n = 1 then
          depart S after
            a := 7
        a := 8
        i := i + 1
      }
      if i = 0 then
        depart R
      c := 3
      m := a + c
      i := i + 1
      if i > n then
        depart R
    }
}
```

Figure 14: A loop before and after Algorithm **PartialPeel** is run

```
Root ← PartialPeel(Root);

PartialPeel(thisNode)
  Replace each child c of thisNode with PartialPeel(c);
  if thisNode is not a loop region node then
    return thisNode;
  body ← thisNode.child.child;
  if (body is not a list node) or (body.regionsDeparted is empty) then
    return thisNode;
  <test, leftOver> ← ExtractTest(body);
  if test is Nothing then
    return thisNode;
  thisNode.child.child ← new ListNode(leftOver, test);
  outerRegion ← new RegionNode;
  outerTest ← recursive copy of test with uses of thisNode
  changed to uses of outerRegion;
  outerRegion.child ← new ListNode(outerTest, thisNode);
  return outerRegion;
end function

ExtractTest(body)
  if (body is Nothing) or (body.first.regionsRepeated is not empty) then
    return <Nothing, body>;
  if body.first.regionsDeparted is not empty then
    return <body.first, body.rest>;
  <test, leftOver> ← ExtractTest(body.rest);
  if test is Nothing then
    return <Nothing, body>;
  return <new ListNode(body.first, test), leftOver>;
end function
```

Figure 15: Algorithm **PartialPeel**

it performs dead code elimination.

Figure 17 shows the effect of Algorithm **CodeMotion**. The same comments regarding execution time and program expansion apply to it as applied to Algorithm **PartialPeel**: while potentially exponential, in practice the execution time should be $O(P)$ (assuming that computing the PDG takes $O(P)$), as should the effect on program size.

To the best of my knowledge, the only CFG algorithm comparable in power to Algorithm **CodeMotion** is the STRICT algorithm of Cytron *et al.* [CLZ86]. It requires dominators. Their paper also discusses a "landing pad" transformation for CFGs that is analogous to partial loop peeling, but they do not give an algorithm for performing it; [Hei90] indicates that such an algorithm is rather complex.

9 Constant Propagation

Constant propagation (CP) discovers values that are constant on any conceivable execution of a program, and propagates them as far forward as possible. This increases the number of expressions and conditionals that can be evaluated at compile-time, thus reducing code size and providing opportunities for other optimizations. In [WZ91], Wegman and Zadeck give an efficient CFG/SSA-based CP algorithm that is as powerful as any known: it finds all constants that can be detected by evaluating all conditional branches having constant operands. I intend to borrow this algorithm, but base it on a PRG rather than a CFG. (Basing it on an AST would be cumbersome due to the lack of links between conditionals and their control dependents.) Since the resulting CP algorithm is nearly identical to Wegman and Zadeck's, I do not present it here.

10 Related Work

Others have noted the merits of using ASTs or other tree-like representations for code optimization, but have only discussed their advantages with regard to traditional data flow analysis problems such as computing live variables. Rosen [Ros77] proposes representing each statement in a program by a graph, and an entire program by an AST-like hierarchy of graphs. This makes data

```
Compute the PDG;
Root ← CodeMotion(Root);
Eliminate any resulting dead code;

CodeMotion(thisNode)
  Replace each child c of thisNode with CodeMotion(c);
  if thisNode is not a loop region node then
    return thisNode;
  body ← thisNode.child.child;
  if body is not a list node then
    return thisNode;
  for each p in thisNode.loopPhiAssignments do
    p.invariant ← false;
  RegionRenames ← new Dictionary;
  <result, leftOver> ← ExtractInvariants(body);
  Change region uses in result according to RegionRenames;
  Compute thisNode.inputVariables;
  return new ListNode(result, leftOver);
end function

ExtractInvariants(thisNode)
  if (thisNode is Nothing) or
  (thisNode.first.regionsRepeated is not empty) or
  (thisNode.first.regionsDeparted is not empty) then
    return <Nothing, body>;
  result ← new ListNode;
  <result.first, thisNode.first> ← Extract(thisNode.first);
  <result.rest, thisNode.rest> ← ExtractInvariants(thisNode.rest);
end function
```

continued ...

Figure 16: Algorithm **CodeMotion**

```
Extract(thisNode)
  case thisNode is
    assignment node:
      thisNode.invariant := IsInvariant(thisNode);
      if not thisNode.invariant then
        return <Nothing, thisNode>;
      return <thisNode, Nothing>;

    if-then node:
      thisNode.invariant := IsInvariant(thisNode);
      if not thisNode.invariant then
        return <Nothing, thisNode>;
      result := Process(thisNode);
      result.condition := copy of thisNode.condition;
      <result.phiAssignments, thisNode.phiAssignments> :=
        Process(thisNode.phiAssignments);

    loop region node:
      for each v in thisNode.inputVariables do
        if not IsInvariant(definition of v) then
          return <Nothing, thisNode>;
      for each p in thisNode.phiAssignments do
        thisNode.invariant := true;
      return <thisNode, Nothing>;

    non-loop region node:
      result := Process(thisNode);
      RegionRenames[thisNode] := result;
      <result.phiAssignments, thisNode.phiAssignments> :=
        Process(thisNode.phiAssignments);

    default:
      result := Process(thisNode);
  end case
  return <result, thisNode>;
end function
continued ...
```

Figure 16: (continued)

```
IsInvariant(thisNode)
  for each x in thisNode.flowSupporters and thisNode.controlSupporters do
    if not x.invariant then
      return false;
  return true;
end function

Process(thisNode)
  result := new node of same type as thisNode;
  for each child component x of thisNode do
    <result.x , thisNode.x> := Extract(thisNode.x);
  return result;
end function
```

Figure 16: (continued)

flow analysis efficient because the graphs for known control structures such as IF-THEN can be "analyzed" when a data flow analysis algorithm is written (at "define-time"); other kinds of graphs must undergo more complex analysis (at compile-time). Sharir [Sha80] proposes parsing a program's CFG into sub-graphs representing IF-THEN-ELSE, WHILE-DO, and so on, and then applying techniques similar to Rosen's. Farrow *et al.* [FKZ76] propose an approach similar to Sharir's, although for a more restricted class of CFGs. The approach most similar in spirit to mine is Babich's "method of attributes" [Bab77], which associates data flow values ("attributes") with each node in the AST, and then iterates over the AST re-computing these values in a manner depending on the node type until they stabilize. Except for that of Farrow *et al.* , these approaches all handle arbitrary GOTOs.

My approach differs from those just described in two respects. First, I make use of the relatively recent notion of SSA form, which simplifies data flow analysis. More importantly, I consider optimizations that depend on the details of control flow, and so do not fit into the standard data flow analysis framework [ASU86], which is *flow insensitive* (branches are assumed to go both ways). Thus Algorithm **CSE**, for example, can find equivalent expressions if they occur under the same control *conditions*, while traditional CSE algorithms can only find them if they occur under the same control

```
region Q {
  i.2 := i.1 + 1
  if i.2 > n.1 then
    depart Q
  region R {
    i.3 := φ(i.2, i.6)
    repeat R after {
      region S {
        if n.1 = 1 then
          depart S after
            a.2 := 7
        a.3 := 8
        i.4 := i.3 + 1
      }
      a.4 := φ(a.2, a.3)
      i.5 := φ(i.3, i.4)
      if i.5 = 0 then
        depart R
      c.2 := 3
      m.2 := a.4 + c.2
      i.6 := i.5 + 1
      if i.6 > n then
        depart R
    }
  }
  m.3 := φ(m.1, m.2)
}
m.4 := φ(m.3)
```

```
region Q {
  i.2 := i.1 + 1
  if i.2 > n.1 then
    depart Q
  region SS {
    if n.1 = 1 then
      depart SS after
        a.2 := 7
    a.3 := 8
  }
  a.4 := φ(a.2, a.3)
  region R {
    repeat R after {
      i.3 := φ(i.2, i.6)
      region S {
        if n.1 = 1 then
          depart S after
            nothing
        i.4 := i.3 + 1
      }
      i.5 := φ(i.3, i.4)
      if i.5 = 0 then
        depart R
      c.2 := 3
      m.2 := a.4 + c.2
      i.6 := i.5 + 1
      if i.6 > n then
        depart R
    }
  }
  m.3 := φ(m.1, m.2)
}
m.4 := φ(m.3)
```

Figure 17: A loop before and after Algorithm **CodeMotion** is run

structure. Conjoining SSA form and control dependence yields optimization algorithms that are able to deal with control more effectively, and so are more powerful than traditional algorithms.

Despite the advantages of ASTs, optimizing compilers have rarely used them. Zelkowitz and Bail [ZB74] discuss how to do CSE and code motion in a structured programming language using an AST. The Bliss compiler [WJW*75] uses an AST for some optimizations and to locate loops, but uses a CFG for most optimizations. The Modula-2 compiler of Powell [Pow84] uses an AST for CSE and code motion.

The code optimizers just mentioned use language-specific ASTs, whereas I propose using **CORTL** ASTs in order to achieve language independence. Furthermore, I intend to use an AST for the entire optimization and code generation process, including register allocation.

11 The Thesis

The bulk of the thesis will consist of six chapters devoted to each of the following topics: an alternative to control flow graphs, data and control dependence, machine-independent optimization algorithms, machine-dependent optimization algorithms, implementation, and measurements.

11.1 An Alternative to Control Flow Graphs

Naturally, the thesis will contain the material presented here on the deficiencies of CFGs and the virtues of **CORTL** ASTs. It will also present an algorithm for translating programs with arbitrary `GOTO`s into **CORTL**.

11.2 Data and Control Dependence

In addition to the material on SSA form and control dependence presented here, the thesis will give a representation-independent definition of SSA form—the definition that Cytron *et al.* give is tied to CFGs—and a proof that the SSA conversion algorithm indeed computes SSA form.

Most data flow problems can be formulated using a *monotone data flow framework* (MDF) [KU76] [KU77] [ZC90], in which a lattice represents possible data flow information, and monotone functions represent the effect that each basic block in the CFG has on the data flow information. It would be interesting to reinterpret the work on MDFs with regard to ASTs for semi-structured languages like **CORTL**. It is probably possible to develop a syntax-directed algorithm for solving MDF problems that is faster than the iterative algorithm used for CFGs. For example, simply traversing the program backwards suffices to solve the "live variable" problem of determining which variables are live at any point: a variable becomes live when a use is encountered, and is dead again after the definition is encountered.

11.3 Machine-Independent Optimization Algorithms

An optimization algorithm is machine-independent if it reduces the amount of intermediate code under the assumption that doing so will usually reduce the execution time of the resulting machine code. (While this assumption is valid most of the time, occasionally it is not: for example, eliminating a common subexpression can increase execution time, since recomputing a value can be faster than reloading it.) Naturally, the thesis will give the machine-independent algorithms presented here. It will also describe several others, such as useless "jump" (`depart` and `repeat`) elimination, dead code elimination, and possibly strength reduction. It will also present a correctness proof for Algorithm **CSE**.

11.4 Machine-Dependent Optimization Algorithms

An optimization algorithm is machine-dependent if it makes important assumptions about the characteristics of the target machine. One such "optimization" is global register allocation, and I hope to give an algorithm for this in the thesis.

There is a close analogy between register allocation and virtual storage management [Dei84]: in optimal page replacement (register allocation) [Bel66], if there are no free pages (registers), then replace (spill) the page (register) that will not be needed again for the longest time. While this analogy works well for local register allocation, using it globally is problematic in a CFG due to the lack of structure [Fre74]. Global register allocation algorithms using graph coloring [CAC*81] [Cha82] have become popular, since they have an elegant conceptual basis. However, as Callahan and Koblenz point out [CK91], these algorithms are insensitive to program structure, and so often make bad decisions as to what and/or where to spill.[4]

I hope to develop a global register allocation algorithm for ASTs using the page replacement analogy that is both simple and capable of making good spill decisions. As evidence for the "simple" part, consider how easy it is to compute the registers that are live at a given point using the algorithm briefly mentioned above. Determining which registers are simultaneously live is a fundamental and—with CFGs—somewhat complex part of constructing the conflict graphs that the graph coloring algorithms operate on.

I also hope to discuss instruction scheduling [BR91] in the thesis. The *hierarchical reduction* technique described in [Lam88] is one indication that using ASTs rather than CFGs can simplify and improve scheduling.

11.5 Implementation

I will implement most of the optimizations mentioned here in the RTL System code optimizer [McC89], and discuss the implementation in the thesis. Of particular interest here will be how well the **CORTL** `if-then` construct maps to the low-level "jump-on-condition" instructions computers possess.

11.6 Measurements

At a minimum, the thesis will compare the CFG-based version of the RTL System to the AST-based one with regard to size, complexity, and speed, since my primary claim is that using ASTs rather than CFGs makes for simpler optimizers. Comparing the effectiveness of the RTL System with other optimizers may be harder: the literature is rather barren of results except in the areas of register allocation and instruction scheduling. Furthermore, only a Smalltalk front-end is currently available for the RTL System, making it difficult to compile standard test suites such as the C-language Stanford integer benchmarks.

[4]The authors propose remedying this deficiency by turning the CFG into a tree!

12 Conclusion

In spite of its lack of explicit structure, the control flow graph representation of programs holds sway in the area of code optimization. However, the reasons for this are at best trifling. With very minor restrictions on program structure, an abstract syntax tree representation can be used instead, leading to algorithms that are simpler and/or more powerful than their graph-based counterparts. The ultimate benefit is simpler—and hence more reliable and easier to maintain—optimizing compilers.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[ASU86] Alfred C. Aho, Ravi Seth, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[Bab77] W. A. Babich. *High level data flow analysis using a parse tree representation of the program.* PhD thesis, University of North Carolina, Chapel Hill, NC, 1977.

[Bel66] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, April 1966.

[BR91] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the SIGPLAN '91 Symposium on Programming Language Design and Implementation, SIGPLAN Notices*, pages 241–255, 1991.

[CAC*81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[CFR*91] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[Cha82] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices*, pages 98–105, 1982.

[CHH89] Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic generation of DAG parallelism. In *Proceedings of the SIGPLAN '89 Symposium on Programming Language Design and Implementation, SIGPLAN Notices*, pages 54–68, 1989.

[CK91] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the SIGPLAN '91 Symposium on Programming Language Design and Implementation, SIGPLAN Notices*, pages 192–203, 1991.

[CLZ86] Ron Cytron, Andy Lowry, and Kenneth Zadeck. Code motion of control structures in high-level languages. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–85, 1986.

[Dei84] Harvey M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley, 1984.

[Dij68] E. W. Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11(3):147–148,538,541, March 1968.

[FKZ76] R. Farrow, K. Kennedy, and L. Zucconi. Graph grammars and global program data flow analysis. In *Proceedings of the Seventeenth Annual IEEE Symposium on Foundations of Computer Science*, pages 42–56, 1976.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Fre74] R. A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM*, 17(11):638–642, November 1974.

[Har85] D. Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 185–194, 1985.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* North-Holland, 1977.

[Hei90] Richard Heintz. *Low Level Optimizations for an Object-Oriented Programming Language.* Master's thesis, University of Illinois at Urbana-Champaign, 1990.

[Hop71] J. E. Hopcroft. An n log n algorithm for minimizing the states of a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196, Academic Press, New York, 1971.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[JM91] Ralph E. Johnson and Carl McConnell. *The RTL System: A Framework for Code Optimization.* Technical Report UIUCDCS-R-9-1698, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1991.

[Knu73] Donald E. Knuth. *Fundamental Algorithms.* Volume 1 of *The Art of Computer Programming*, Addison-Wesley, 1973.

[Knu74] Donald Knuth. Structured programming with goto statements. *ACM Computing Surveys*, 6(4):261–301, December 1974.

[Kra88] David A. Kranz. *ORBIT: An Optimizing Compiler for Scheme.* PhD thesis, Yale University, 1988.

[KU76] J. B. Kam and J. D Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, 1976.

[KU77] J. B. Kam and J. D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–318, 1977.

[KZ77] Ken Kennedy and Linda Zucconi. Applications of a graph grammar for program control flow analysis. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 72–85, 1977.

[Lam88] Monica Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88*

Symposium on Programming Language Design and Implementation, SIGPLAN Notices, pages 318–328, 1988.

[LT79] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flow graph. *ACM Transactions on Programming Languages and Systems*, 1:121–141, 1979.

[McC89] Carl McConnell. *An Object-Oriented Code Optimizer Using Static Single Assignment Form*. Master's thesis, University of Illinois at Urbana-Champaign, 1989.

[PKT73] W. W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat, and exit statements. *Communications of the ACM*, 16(8):503–512, 1973.

[Pow84] Michael L. Powell. A portable optimizing compiler for Modula-2. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, pages 310–318, 1984.

[Ros77] Barry K. Rosen. High-level data flow analysis. *Communications of the ACM*, 20(10):712–724, October 1977.

[Sha80] M. Sharir. Structural analysis: a new approach to flow analysis in optimizing compilers. *Computer Languages*, 5:141–153, 1980.

[Ste78] Guy L. Steele. *RABBIT: A Compiler for Scheme*. Technical Report, Massachusetts Institute of Technology, May 1978.

[Weg73] Eberhard Wegner. Tree-structured programs. *Communications of the ACM*, 16(11):704–705, November 1973.

[WJW*75] W. A. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke. *The Design of an Optimizing Compiler*. American Elsevier, 1975.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[YHR89] Wuu Yang, Susan Horwitz, and Thomas Reps. *Detecting Program Components with Equivalent Behaviors.* Technical Report 840, Computer Sciences Department, University of Wisconsin-Madison, April 1989.

[ZB74] M. V. Zelkowitz and W. G. Bail. Optimization of structured programs. *Software—Practice and Experience*, 4(1):51–57, January 1974.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1990.