Team 4 (Crystal)

Project Proposal

Constant Propagation and Dead Code Elimination

김도형 (2018-22640) 정재환 (2018-26716) Alena Kazakova (2017-21749)

1. Project Objective

- ➔ Develop a Dataflow analysis pass.
- ➔ Implement constant propagation and dead code elimination based on dataflow analysis.
- ➔ Extend implementation with interprocedural dataflow analysis.
- ➔ Evaluate implementation.

2. Basic Approach

2.1 Dataflow Analysis

Based on the given source, the following precondition has been met:

AST (given with blackbox) → TAC (ir.cpp) → *.s code (backend.cpp)

Among these outputs, TAC will be handled by the crystalCFG*[1], which will generate CFG. Once the CFG is created, we will use the iterative dataflow analysis to analyze relationships between instructions.

2.2 Constant Propagation

There are several constant propagation techniques. We chose the global constant propagation method, which is an optimization task that requires dataflow analysis. In order to deal with conditional branches, we chose a simple well-known global dataflow analysis algorithm (detailed description below).

Value-Interpretation Table

Value	**Interpretation**
$\perp$	This statement never executes
c	X = constant c
T	X is not a constant

For each statement s (one line of code), we compute information about the value of x immediately before and after s:

$C(s, x, in)$ = value of x before s
$C(s, x, out)$ = value of x after s

[1] crystalCFG* : implemented source for generating CFG based on given AST

We define a transfer function that transfers information from one statement to another. In the following rules, let statement *s* have immediate predecessor statements $p_1, ..., p_n$.

Rule 1 : if $C(p_i, x, out) = \bot$, for any *i*,	$C(s, x, in) = T$
Rule 2 : if $C(p_i, x, out) = c$ & $C(p_j, x, out) = d$,	$C(s, x, in) = T$
Rule 3 : if $C(p_i, x, out) = c$ or $\bot$ for all *i*,	$C(s, x, in) = c$
Rule 4 : if $C(p_i, x, out) = \bot$ for all *i*,	$C(s, x, in) = \bot$
Rule 5 : if $C(s, x, in) = \bot$,	$C(s, x, out) = \bot$
Rule 6 : if *c* is a constant,	$C(x := c, x, out) = c$
Rule 7 : $C(x := f(...), x, out) = T$	
Rule 8 : if $x <> y$,	$C(y := ..., x, out) = C(y := ..., x, in)$

Algorithm:

```
1. For every entry s to the program, set $C(s, x, in) = T$
2. Set $C(s, x, in) = C(s, x, out) = \bot$ everywhere else
3. Repeat until all points satisfy rules 1-8:
        Pick s not satisfying 1-8 and update using the appropriate rule.
```

2.3 Dead Code Elimination

To eliminate dead code, the liveness information (Boolean property) of statement is required.
There are few rules to check the liveness condition:

Rule 1 : $L(p, x, out) = \vee \{ L(s, x, in) \mid s$ a successor of $p\}$
Rule 2 : $L(s, x, in)$ = true, if *s* refers to *x* on the rhs
Rule 3 : $L(x := e, x, in)$ = false, if *e* does not refer to *x*
Rule 4 : $L(s, x, in) = L(s, x, out)$, if *s* does not refer to *x*
(*p*: predecessor, *s*: statement, *L*: liveness, *e*: constant)

Algorithm:

```
1. Let all L(...) = false initially
2. Repeat until all statements satisfy rules 1-4
```

3. Extension

The basic approach can be extended by replacing intraprocedural dataflow analysis with interprocedural dataflow analysis for previously proposed constant propagation and dead code elimination algorithms. Interprocedural analysis starts with the statically observable targets, and then incorporates the new edges into the call graph as more targets are discovered until convergence is reached.

There are two approaches to interprocedural analysis: context-insensitive (simple but inaccurate) and context-sensitive (has choice in precision). Context-sensitive analysis can be cloning-based (applies context-insensitive analysis to a cloned graph for each unique context of interest) or summary-based (avoids reanalyzing a procedure's body at every call site that may invoke the procedure). In case we do not come up with our own idea, we are inclining to implement a context-sensitive summary-based analysis as it can provide more efficient and precise results.

4. Implementation and Evaluation Plan

The implementation consists of two main parts: basic approach and extension. We start from the dataflow pass construction, followed by implementation of constant propagation and dead code elimination. After verifying the basic implementation, we will proceed to extending our optimization with interprocedural dataflow analysis for constant propagation and dead code elimination.

The most important step of the evaluation is verifying the correctness of the compiler's output. Then, we shall evaluate the execution time of our optimized implementation vs SnuPL/1 using provided test programs. We shall write our own test cases if necessary. Evaluation will be performed by using C standard library time function. For diversity of the analysis, we shall also compare the size of assembly code files. Lastly, we shall conclude the effect of our optimization techniques.

Final Project Timeline[2]

	9-4	10-1	10-2	10-3	10-4	11-1	11-2	11-3	11-4	12-1	12-2
Dataflow Construction	🟥	🟥	🟥	🟥							
Implementation			🟥	🟥	🟪	🟥					
Extension					🟪	🟥	🟥	🟥			
Debug & Modification		🟥	🟥	🟥	🟪	🟥	🟥	🟥	🟥		
Evaluation									🟥	🟥	
Final Report, Presentation										🟥	🟥

6. References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
2. Andrew W. Appel and Maia Ginsburg. 2004. Modern Compiler Implementation in C. Cambridge University Press, New York, NY, USA.
3. Steven S. Muchnick. 1998. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
4. Alex Aiken. Compilers. Stanford University. Retrieved from http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=Compilers

[2] The schedule is flexible depending on our progress and individual team members' schedule.