Automating Derivation of Incremental Programs *

Yuchen Zhang Yanhong A. Liu
Computer Science Department, Indiana University, Bloomington, IN 47405
{yuczhang, liu}@cs.indiana.edu

Given a program f and an input change operation $\oplus$, an *incremental program* f' computes the value of $f(x \oplus y)$ efficiently by making use of the value of $f(x)$. Liu and Teitelbaum proposed a systematic approach [1] for deriving incremental programs written in a first order functional programming language. This work aims to automate the derivation of f' from f and $\oplus$ based on a semi-automatic implementation – CACHET [2]. The derivation of incremental program shares the same underlying principle with finite differencing and a number of other program optimizations. It is crucial for optimizing programs in high-level language.

Derivation of incremental programs

Liu and Teitelbaum's approach has three major components: simplification, replacement using cached results, and introduction of incremental functions. The derivation begins with $f(x \oplus y)$. It examines subcomputations recursively in applicative and left-to-right order and applies simplifications and replacements with retrievals from cached results. If a function application can't be replaced by a retrieval, then replace it with an incremental version: if it is an instance of an incremental version alreadyintroduced, then simply replace it; otherwise, introduce a corresponding incremental version for the unfolded application and recursively apply simplifications and replacements on the new function. Liu's semi-automatic implementation has automatic rules for introducing incremental functions, local simplification and replacement, but the entire derivation needs manual invocation of appropriate rules at each step.

Applicative-order rewrite

Automation of CACHET continued to use the Synthesizer Generator [3]. To automate the derivation, there are three major challenges. First, sequencing the transformation while maintaining derived information at appropriate program points. Second, automating equality reasoning needed for local replacement. Third, ensuring termination.

We implemented three modules corresponding to the three major components of the derivation algorithm. A main module traverses a program in applicative order. To make the automated derivation efficient, transformation rules are grouped according to the program constructs they apply to so as to reduce rule mismatch.

There are two kinds of equality reasoning: one for data constructions, the other for arithmatic operation. For the first kind, equality reasoning is based on the relation between a constructor and its corresponding selectors. For the second kind, equality reasoning uses the simplification of arithmatic formula. For example, $x + 1 - 1 = x$. Our general algorithm achieves this by grouping constants and organizing variables. We plan to switch to Omega [4] for the second kind of reasoning.

Introducing new incremental function involving unfolding is the source of non-termination. We set a bound on the number of incremental versions that can be introduced for a function. Thus, the transformation terminates either when the derivation completes or such bounds are reached.

Applications

Incremental computation has applications in optimizing compilers, transformational programming, interactive programming environments, etc. The derivation described here allows only the use of the return value of f. However, when f is extended to return also additional information, our derivation will yield a program that uses and maintains all such information. For example, given a binomial coefficient program that returns all the intermediate results of function calls:

$bino(i, j)$ *where* $0 \leq \mathrm{j} \leq \mathrm{i}$
$=$ **if** $j = 0$ **or** $j = i$ **then** $< 1 >$
  else let $v1 = bino(i-1, j-1)$ **in**
    let $v2 = bino(i-1, j)$ **in**
    $< 1st(v1) + 1st(v2), v1, v2 >$

For input change operation $\oplus$: $\langle i', j' \rangle = \langle i+1, j \rangle$, our automated system produces the following incremental program, which can be used as the body of a repeated computation that forms a dynamic programming solution. The speedup achieved is from exponential time to polynomial time.

$bino'(i, j, r)$
$=$ **if** $j = 0$ **or** $j = i + 1$ **then** $< 1 >$
  else if $j = i$ **then**
    let $v1 = bino'(i-1, j-1, < 1 >)$ **in**
    $< 1st(v1) + 1, v1, < 1 >>$
  else let $v1 = bino'(i-1, j-1, 2nd(r))$ **in**
    let $v2 = r$ **in**
    $< 1st(v1) + 1st(v2), v1, v2 >$

We have applied the system on a number of small examples, including programs for sorting, other list operations, and simple matrix computations. We translated both the original and incremental programs in our subject language to Scheme and compared the running times. The observed speedups confirmed our asymptotic analysis.

We plan to further modularize the system and improve the equality reasoning and other analyses used.

*This work is supported by a Motorola University Partnership in Research Grant and NSF grant $CCR - 9711253$. Yuchen Zhang is a student recipient of Motorola University Partnerships in Research Grant.

References

[1] Yanhong A. Liu and Tim Teitelbaum. Systematic Derivation of Incremental Programs. *Science of Copmuter Programming*, 24(1):1-39, Feb. 1995.
[2] Yanhong A. Liu. CACHET: An Interactive, Incremental-Attribution-Based Program Transformation System for Deriving Incremental Programs. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*. November 1995. IEEE Computer Society Press.
[3] William Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM (CACM)*, 31(8), August 1992.
[4] The Synthesizer Generator Reference Manual, Release 5.0. GrammaTech, Inc. Ithaca, New York, 1996.