Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A systematic incrementalization technique and its application to hardware design

Steven D. Johnson[*], **Yanhong A. Liu**[**], **Yuchen Zhang**[***]

Indiana University Computer Science Department, e-mail: sjohnson@cs.indiana.edu

The date of receipt and acceptance will be inserted by the editor

Abstract. A systematic transformation method based on *incrementalization* and value *caching* generalizes a broad family of program optimizations. It yields significant performance improvements in many program classes, including iterative schemes that characterize hardware specifications. *CACHET* is an interactive incrementalization tool. Although incrementalization is highly structured and automatable, better results are obtained through interaction, where the main task is to guide term rewriting based on data specific identities. Incrementalization specialized to iteration corresponds to *strength reduction*, a familiar program refinement technique. This correspondence is illustrated by the derivation of a hardware-efficient nonrestoring square-root algorithm, which has also served as an example of theorem prover based implementation verification. One goal of this study is to explore how ingenious design insights are discovered and applied in contrasting formal systems, as reflected in their supporting tools.

Key words: Formal methods – hardware verification – design derivation – formal synthesis – transformational programming – floating point operations

* Supported, in part, by the National Science Foundation under grant MIP-9601358.

** Supported in part by the National Science Foundation under grant CCR-9711253, the Office of Naval Research under grant N00014-99-1-0132, and Motorola Inc. under a Motorola University Partnership in Research Grant.

*** Student recipient of a Motorola University Partnership in Research Grant

1 Introduction

The transformation technique described in this paper will be familiar to all programmers and digital engineers. It centers on *incremental* computation, the exploitation of partial results to more efficiently calculate new results. We present here a general method for performing such optimizations and a tool called *CACHET* for systematically applying that method.

We introduce *incrementalization* through a series of small examples, culminating with the derivation of a *nonrestoring integer square root* implementation originally verified in Nuprl by O'Leary, Leeser, Hickey, and Aagaard [17]. That, too, was a tutorial illustration of formalized reasoning in a hardware design context. One purpose of this study is to explore how the critical insights needed to improve a design are discovered and applied in a given reasoning framework, as reflected in its tools. In particular, we are interested in contrasting *deductive* verification, in which the design process formalized as a proof, with *derivational* verification, in which the design process is formalized as a sequence of equivalence preserving refinements.

This is not a question of which style is "better," but of gaining understanding about how intelligent judgments are made so that they can be better facilitated in an integrated reasoning environment. The ultimate goal is an environment incorporating a broad variety of reasoning systems, both automatic and interactive. In order to successfully reach that goal, we need a clearer picture of how humans interact, and especially how creative judgement arises.

The sqrt example of Section 4 is relatively small, but otherwise it is representative of real designs in signal

```
n, i, m := input, (l − 2), 2^(l−1);
while i ≥ 0 do
   p := n − m^2;
   if p > 0 then
      m := m + 2^i
   else if p < 0 then
      m := m − 2^i;
   i := i − 1;
output := m
```

```
p, v, w := input, 0, 2^(2(l−1));
while (w ≥ 1) do
   if p > 0 then
      p, v, w := p − v − w; v/2 + w, w/4
   else if p < 0 then
      p, v, w := p + v − w, v/2 − w, w/4
   else
      v, w := v/2, w/4;
output := v
```

Fig. 1. Specification and implementation of nonrestoring sqrt

processing, arithmetic units, microprocessor pipelines, and so on. Behavioral forms of the sqrt specification and implementation, expressed in a statement oriented syntax, are shown in Figure 1. The essence of implementation verification, the key insights, are algebraic identities—in this case, laws of arithmetic but generally, equational laws of a type structure over which the specification and implementation are expressed.

The implementation in Figure 1 is readily reduced to hardware. The *DDD transformation system* is an interactive tool for formally deriving and manipulating hardware architecture descriptions from behavioral specifications [6,2]. O'Leary, et. al. include a hardware description in their implementation proof, using an ML variant to express architectural structures. Thus, a single reasoning tool, Nuprl, analyzes both behavioral and architectural expressions. The derivation study also uses dialects of functional notation to represent behavior and architecture, but CACHET applies only to behavioral forms. DDD translates control oriented expressions to architecture oriented expressions and reduces the latter to boolean systems.

It is shown in Section 4 that, beyond some linear inequalities, judicious applications of distributivity, associativity, the identity

$$(x + y)^2 = x^2 + 2xy + y^2$$

suffice to support an implementation proof at the integer level. This observation holds whether the argument is deductive or derivational. However, "judicious application" implies that the design agent not only has the insight to tactically apply algebraic identities, but also understands the logical context, that is, the overall strategy and form of the proof.. We are interested in contrasting how insights are discovered, visualized, and applied in various reasoning frameworks.

This paper has two main goals. The first is to introduce the analyses and constructions that, together, comprise incrementalization. We begin with a small motivating example relating it to *strength reduction*, a classical program transformation technique. Two examples follow to illustrate generality, in particular, extending the idea of strength reduction to nonlinear recursion patterns. In software, application of incrementalization has been shown to yield dramatic asymptotic performance improvements through recursion removal [12].

Loop strength reduction is an important special case, especially as it applies to hardware design. The second goal is to illustrate how incrementalization specializes to the iterations common in hardware specification. In this context an incrementalization tool, like CACHET introduced in Section 5, facilitates the interplay of designer insight with formal manipulation.

1.1 Background

The core approach to *incrementalization* is described by Liu in her dissertation [13,14]. An incrementalization tool, *CACHET* is the focus of [9]. Subsequently, extensions to the basic approach have addressed *caching*, or maintaining partial results in auxiliary variables [11,12]. Caching uses an on-line dependence analysis to prune unneeded accumulators. In [10], Liu outlines the steps of a systematic, semi-automatable incrementalization process, including a brief presentation of the sqrt derivation detailed here in Section 4. Incremental computation is involved in a broad family of optimization techniques, surveyed in [13].

Design derivation refers to a formalized design process in which a creative agent interacts with a reasoning tool to transform a specification into a correct implementation. Johnson, Bose, Miner, and others have investigated an integrated framework for formalized design in which a derivational tool, *DDD*, interacts with a theorem prover. It is demonstrated in [2,1] that such a heterogeneous framework reduces the effort of verifying a microprocessor implementation. In [7,15], Miner explores a more tightly coupled relationship between a derivational and deductive formalisms. These studies raise basic

questions about the character of interaction as reflected in the analysis tools.

The past few years have seen increasing attention paid to term-level reasoning in hardware verification. Validity checking with uninterpreted function symbols (e.g. [8]) is a way to increase the power of model checking and adapt to data path aspects. At the same time, theorem proving approaches have repeatedly demonstrated that the essence of hardware verification lies in equational reasoning performed in the complicated logical context of an implementation proof. Moore's description of a *symbolic spreadsheet* [16] reflects this insight. Greve's analysis of the JEM1 microprocessor, and other similar case studies, explore interactive verification centering on symbolic simulation (i.e. function expansion) and term simplification [4].

In 1993, Windley, Leeser, and Aagard pointed out that numerous hardware verifications have been found to follow a common proof plan [21]. Incrementalization might be seen as a "super duper" derivation tactic, but one that is applicable to a much broader range of specification classes than just hardware.

1.2 Strength reduction

Incrementalization generalizes a basic programming technique found in virtually all approaches to program refinement, however formal. The illustration below comes from an undergraduate textbook written in 1978 [20], which credits Dijkstra for the phrase "strength reduction" [3]. We use the notation $\{P\}\ S\ \{Q\}$ to express partial correctness, "If precondition P holds then execution of program S establishes postcondition Q."

We want an algorithm to compute the integer square root of an input x; that is, an S such that

$$\{0 \le x\}\ S\ \{z^2 \le x < (z+1)^2\}$$

The *and* in the postcondition suggests a loop, with one conjunct serving as the loop's test and the other the loop invariant [5]:

$$\begin{array}{l} z := 0; \\ \textbf{while}\ x \ge (z+1)^2\ \textbf{do} \\ \quad \{z^2 \le x\} \\ \quad z := z+1 \\ \{z^2 \le x < (z+1)^2\} \end{array}$$

To get rid of the expensive term $(z+1)^2$, we can introduce an auxiliary variable u to hold this value. The invariant becomes $\text{INV}_1 \equiv z^2 \le x \ and\ u = (z+1)^2$ and the loop is be adapted to maintain the stronger condition. Let u' and v' denote the values of u and v after the next loop iteration. The analyses (simultaneous in general)

$$z' = z+1 \qquad \begin{array}{l} u' = (z'+1)^2 \\ \quad \stackrel{?}{=} {z'}^2 + 2z' + 1 \\ \quad = (z+1)^2 + 2z' + 1 \\ \quad \stackrel{!}{=} u + 2z' + 1 \end{array}$$

eliminates the squaring operation. Of course, u must be properly initialized.

$$\begin{array}{l} (z,u) := \langle 0, 1\rangle; \\ \textbf{while}\ x \ge u\ \textbf{do} \\ \quad \{\text{INV}_1\} \\ \quad z := z+1; \\ \quad u := u + 2z + 1 \end{array}$$

There are four discussion points.

First, the derivation of u' exploits the algebraic identity, $(X+1)^2 = X^2 + 2X + 1$, at the third step. In general, we cannot expect such insight be fully automated because term equivalence is undecidable in some structures, including arithmetic. From here on, we refer to the application of algebraic laws as an *exercise of judgment*, presumably by an ingenious agent. We indicate points of judgment with the symbol '$\stackrel{!?}{=}$'. However, even if *no* such interventions take place, programs are often improved by reusing intermediate results, as optimizing compilers commonly do. Incomplete or specialized equational reasoning can improve the result still more, even if it can't always achieve the optimum automatically.

Second, while incrementalization generally has the goal of exploiting partial results to eliminate expensive operations, the measure of expense depends on the target technology. In the program above, if we regarded "$2z$" as expensive, we could eliminate it by introducing a second auxiliary variable variable, to maintain $\{w = 2z\}$. The analysis $w' = 2z' = 2(z+1) \stackrel{!?}{=} 2z + 2 = w + 2$ shows we can convert *multiply-by-two* to *add-two*, provided we can see to apply the distributive law. Obviously, this is not an improvement for hardware, and it may or may not be for software, depending on the compiler.

Third, while we can certainly argue that the elimination of $(z+1)^2$ improves the program, it is still linear in the magnitude of its input. Faster convergence requires a better algorithm, such as the nonrestoring sqrt in Section 4.

Finally, loop invariants are a formal device for declaring *intent*. They are used here for the more limited purpose of reasoning about incremental computation. The underlying optimization tactic is *loop unrolling* (or *unfold/fold* transformation). If we know the optimization technique being applied, using the more general method

of inductive assertions might be considered overkill. One measure of a good method is how broadly it can be applied, but this is not necessarily the case for tools, where specialization can result in a higher degree of automation and more perspicuous notation.

2 Systematic incrementalization

In this section we survey the general approach to incrementalization. Programs are represented as *recursion equations*, that is, systems of first-order function definitions. Each defining expression is a conditional whose branches are either simple *terms* or *expressions* involving recursive calls to the defined functions. Terms come from a ground type, or algebraic structure whose specification includes a set of equational laws. "Term level reasoning" refers to derivations according to these laws. Decidability depends on the decision problem for the given structure. All of the examples in this article involve arithmetic operations with the usual laws of algebra. However, the techniques apply to any abstract data type. As in a theorem prover, the underlying identities are input to the system and are not built in.

Our program notation is conventional with one exception: formal parameters may include nested identifiers and aliasing. For example, the phrase

$$\textbf{let } r = (r_1, r_2) = \mathcal{E} \textbf{ in} \ldots$$

binds the identifier r to the value of expression $\mathcal{E}$ and also declares this value to be a pair whose first and second elements are identified by r_1 and r_2, respectively. We shall use these forms only for simple destructuring.

In the case that all the functions defined in a system are tail-recursive, we have the equivalent of a sequential program. In these cases, we may use the **while**-program notation, as in Section 1, for those who are more comfortable with that form of expression.

The incrementalization method is actually an interplay between two kinds of function extension, *incrementing* and *caching*, as indicated in Figure 2. A function $F\colon W \to V$ stands for the specification to be transformed. Let $\oplus\colon W \to W$ denote a *state mutator*[2], or nonrecursive combination of elementary operations on F's input domain. The *incrementalization of F with respect to $\oplus$* is a function that computes $F(\oplus(w))$ given the value of $F(w)$. That is, $F'\colon (W \times V) \to V$ has the property that

$$F'(w, F(w)) = F(\oplus(w))$$

We want to describe how $F(w)$ is used in calculating the final result. For example, if F returns a data structure; then incrementalization involves analyzing how components of $F(w)$ are reused in creating the object $F(\oplus(w))$.

Caching extends a function to return partial results. $F\colon W \to V$ is extended to $\overline{F}\colon W \to V^k$; that is, $\overline{F}(w) = \langle v_1, v_2, \ldots, v_k \rangle$ with the proviso that $v_1 = F(w)$. The remaining v_i are intermediate values, accumulated in the computation of v_1. Of course, the idea is to cache only those values that will be useful later. This determination requires a dependence analysis similar to *strictness analysis* [12,18]. Thus, in incrementalization, what we are really after is $\overline{F}'$, the incremented caching extension of F.

F's increment represents just one step, or branch, of the computation. It remains to *incorporate* this step in the original program. In functional expressions, incrementing is analogous to unfolding and incorporation to folding, as we shall see in the examples that follow.

2.1 Example 1 – Application to recursion

In Figure 3, we begin with the "*Fibonacci*" scheme, whose recursion pattern is quadratic, incrementalized with respect to $x + 1$; that is, F' computes $F(x + 1)$ given $F(x)$. We would later apply this analysis to the recursive call "$F(x - 1)$" in order to get a way of computing $F(x)$ given $F(x - 1)$. The simple increment (Fig. 3, top right) doesn't get us very far, because, were it incorporated in the original, the result would still be a quadratic recursion:

$$\begin{array}{rcl} F(x) & \stackrel{\diamond}{=} & \textbf{if } x \le 1 \\ & & \quad \textbf{then } 1 \\ & & \quad \textbf{else } \boxed{F'(x-1, F(x-2))} \\ & = & \textbf{if } x \le 1 \\ & & \quad \textbf{then } 1 \\ & & \quad \textbf{else } \boxed{\begin{array}{l} \textbf{let } r = F(x-1) \\ \textbf{in if } x \le 1 \textbf{ then } 1 \\ \quad \textbf{else if } x = 2 \textbf{ then } 2 \\ \quad \textbf{else } r + F(x-2) \end{array}} \end{array}$$

A *caching* version accumulates not only $F(x)$ but also all relevant intermediate values. Assuming addition is "cheap," there are two intermediate values,

$$\overline{F}(x) = \langle F(x), \overline{F}(x-1), \overline{F}(x-2) \rangle$$

leading to the definition in Figure 3, lower left. In forming the increment of $\overline{F}$, the goal is to *prune* unneeded or redundant intermediate values. Since

$$\overline{F}(x-1) = \langle F(x-1), \overline{F}(x-2), \overline{F}(x-3) \rangle$$

the call to $\overline{F}(x-2)$ it can be replaced by u_2:

$$\begin{array}{l} \ldots \textbf{ let } u = (u_1, u_2, u_3) = \overline{F}(x-1) \textbf{ in} \\ \quad \textbf{let } v = (v_1, v_2, v_3) = u_2 \textbf{ in} \\ \quad \langle u_1 + v_1, u, v \rangle \end{array}$$

$\oplus: W \to W$	*original*	*increment*
original	$F: W \to V$	$F': W \times V \to V$ $\models F'(w, F(w)) = F(\oplus(w))$
caching	$\overline{F}: W \to V^n$ $\models \overline{F}(w) = \langle F(w), v_2, \ldots \rangle$	$\overline{F}': W \times V^n \to V^n$ $\models \overline{F}'(w, y, \overline{F}(w)) = \overline{F}(\oplus(w))$

Fig. 2. Components of incrementalization.

These are not defining equations, but identities relating cached, incrementalized, and cached-incrementalized variants of F.

Furthermore, the value of interest, $u_1 + v_1$, does not depend on subcomponents u_3, v_2, or v_3. An automatic analysis of transitive dependence, similar to strictness analysis [18], makes this determination [12]. With the irrelevant terms pruned from the computation, we get

$$\begin{array}{l} \ldots \textbf{let } u = (u_1, u_2, -) = \overline{F}(x-1) \textbf{ in} \\ \quad \textbf{let } v = (v_1, -, -) = u_2 \textbf{ in} \\ \quad \langle u_1 + v_1, u_1, - \rangle \end{array}$$

The cached-incrementalized function $\overline{F}'$ in Figure 3 also prunes the triples to pairs. The structure is now linear, not tree-like, a positive development.

Incorporating these optimizations into the original scheme and doing some elementary transformations, we obtain

$$\begin{array}{l} \overline{F}(x+1) = \textbf{if } x \le 0 \textbf{ then } \langle 1 \rangle \\ \qquad\qquad \textbf{else if } x = 1 \textbf{ then } \langle 2, 1 \rangle \\ \qquad\qquad \textbf{else let } u = (u_1, u_2) = \overline{F}(x) \\ \qquad\qquad\qquad \textbf{in } \langle u_1 + u_2, u_1 \rangle \end{array}$$

Letting $z \equiv x + 1$, we can rewrite this as

$$\begin{array}{l} \overline{F}(z) = \textbf{if } z \le 1 \textbf{ then } \langle 1 \rangle \\ \qquad\qquad \textbf{else if } z = 2 \textbf{ then } \langle 2, 1 \rangle \\ \qquad\qquad \textbf{else let } u = (u_1, u_2) = \overline{F}(z-1) \\ \qquad\qquad\qquad \textbf{in } \langle u_1 + u_2, u_1 \rangle \end{array}$$

The important outcome is that the result is a linear recursion, derived, not proven (although it certainly could be); verification is subsumed by the pruning analysis. It requires associativity to obtain the iterative version of *Fibonacci*. This, too, is derivable using (for example) techniques originated by Wand [19].

$$\begin{array}{l} F(z, 1, 2) \textbf{ where} \\ F(z, u, v) \stackrel{\diamond}{=} \\ \quad \textbf{if } z \le 0 \textbf{ then } u \\ \quad \textbf{else } F(z-1, v, u+v) \end{array}$$

For readers who prefer a statement-oriented form, this function is expressed as

$$\begin{array}{l} \{z = x_0\} \\ u, v := \langle 1, 2 \rangle; \\ \textbf{while } z > 0 \textbf{ do} \\ \quad (z, u, v) := \langle z - 1, v, u + v \rangle; \\ \{v = F(x_0)\} \end{array}$$

Liu, Stoller and Teitelbaum present a number of algorithms whose performance is significantly improved by incrementalization. In most cases, the improvement is a consequence of recursion removal resulting from caching and pruning nonlinear data structures [12]. These examples and others demonstrate that the incrementalization subsumes a broad family of optimization and refinement techniques [10].

In summary *incrementalization* has three main phases: *caching* partial values, which entails *pruning* irrelevant subcomputations; *incrementing* with respect to a state mutator; and *incorporating* the result in the original computation.

3 Specialization to strength reduction

Applying incrementalization to iterative systems corresponds to performing strength reduction on loops. Since we are dealing with loops, we will sometimes use a simple statement oriented language[1].

$\oplus(x) \stackrel{\diamond}{=} x+1$	*original*	*increment*
original	$F(x) \stackrel{\diamond}{=}$ **if** $x \leq 1$ **then** 1 **else** $F(x-1)+F(x-2)$	$F'(x,r) \stackrel{\diamond}{=}$ **if** $x \leq 0$ **then** 1 **else if** $x = 1$ **then** 2 **else** $r + F(x-1)$ $\models F'(x, F(x)) = F(x+1)$
caching pruned	$\overline{F}(x) \stackrel{\diamond}{=}$ **if** $x \leq 1$ **then** $\langle 1, -, - \rangle$ **else let** $(u_1,u_2,u_3) = \overline{F}(x-1)$, $(v_1,v_2,v_3) = \overline{F}(x-2)$ **in** $\langle u_1+v_1, u, v\rangle$ $\models \overline{F}(x) = \langle F(x), \overline{F}(x-1), \overline{F}(x-2)\rangle$	$\overline{F'}(x,r) \stackrel{\diamond}{=}$ **if** $x \leq 0$ **then** $\langle 1, - \rangle$ **else if** $x = 1$ **then** $\langle 2, \langle 1, -\rangle\rangle$ **else let** $(u_1,u_2) = r$ **in** **let** $(v_1,v_2) = u_2$ **in** $\langle u_1+v_1, \langle r_1, -\rangle\rangle$ $\models \overline{F}'(x, \overline{F}(x)) = \overline{F}(x+1)$

Fig. 3. Cached and incremented versions of *Fibonacci*

3.1 Example 2 – application to a loop

As a first illustration, consider the *integer division* program below.

$$\begin{array}{l} \{x = A \ and \ y = B\} \\ z := 1; \\ \textbf{while } y \times z < x \textbf{ do} \\ \quad z := z + 1; \\ \{z - 1 = A \div B\} \end{array}$$

Let us move the expensive operation, '$\times$' into the loop:

$$\begin{array}{ll} & \textbf{while } w < x \textbf{ do} \\ \ldots & \quad z := z + 1; \\ & \quad w := y \times z \ \ldots \end{array}$$

The function to increment is the body of the loop, which computes next-state values for y, z, and w. Incrementing with respect to the *same* function, it is the equivalent of unrolling the loop. That is,

$$\begin{array}{rcl} F(y,z,w) & \stackrel{\diamond}{=} & \langle y, z+1, y \times z\rangle \\ \oplus(y,z,w) & \stackrel{\diamond}{=} & \langle y, z+1, y \times z\rangle \end{array}$$

In this example there are no values to be cached. Let $r = (r_y, r_z, r_w) = F(y,z,w)$. Then

$$\begin{array}{rcl} F(\oplus(y,z,w)) & = & \langle y, (z+1)+1, y \times (z+1)\rangle \\ & \stackrel{!?}{=} & \langle y, (z+1)+1, (y \times z) + y\rangle \\ & = & \langle r_y, r_z + 1, r_w + r_y\rangle \end{array}$$

Use of distributivity in the second step is marked as an exercise of judgment—though it might well be automatic—in order to illustrate one point where judgement is typically required in incrementalization. We can now form the incrementalized loop, removing multiplication in favor of addition.

$$\begin{array}{l} \{x = A \ and \ y = B\} \\ \textbf{if } y < x \\ \quad \textbf{then } z := 1 \\ \quad \textbf{else } z := 2; \\ \qquad w := y \times 2; \\ \qquad \textbf{while } w < x \textbf{ do} \\ \qquad\quad z := z + 1; \\ \qquad\quad w := w + y \\ \{z = A \div B\} \end{array}$$

Elementary transformations are used to fold the initialization back into the loop, as shown in Figure 4.

3.2 Example 3

Let us return momentarily to the the naive sqrt from Section 1.2. With the expensive squaring operation moved from the loop's test to its body, the function to increment is

$$S(x,z,u) = \langle x, z+1, ((z+1)+1)^2)\rangle$$

Again, because we are dealing with a loop, we want $\oplus = S$. Assume $r = (r_x, r_z, r_u) = S(x,z,u)$, and for clarity let us write $\hat{z}$ in place of "$z+1$". Then

$$(r_x, r_z, r_u) = \langle x, \hat{z}, (\hat{z}+1)^2\rangle \qquad (1)$$

$\oplus = F$	*original*	*increment*

original	$z := 1;$
	$w := y \times 1;$
	while $w < x$ **do**
	  $z := z + 1;$
	  $w := y \times z;$
		$\{x = A \text{ and } y = B\}//z := 1;$
		$w := y;$
		while $w < x$**do**
		  $z := z + 1;$
		  $w := w + y$

caching	*not used*	*not used*

Fig. 4. Incrementalization of integer division.

and hence,

$$
\begin{array}{ll}
S(\oplus(x,z,u)) & \\
= S(x,\hat{z},(\hat{z}+1)^2) & (\oplus) \\
= \langle x,\hat{z}+1,((\hat{z}+1)+1)^2\ \rangle & (S) \\
\stackrel{!?}{=} \langle r_x,r_z+1,((\hat{z}+1)+1)^2\ \rangle & \text{(eq. 1)} \\
\stackrel{!?}{=} \langle r_x,r_z+1,(\hat{z}+1)^2+2(\hat{z}+1)+1\ \rangle & \\
= \langle r_x,r_z+1,r_u+2(\hat{z}+1)+1\ \rangle & \text{(eq. 1)} \\
= \langle r_x,r_z+1,r_u+2(r_z+1)+1\ \rangle & \text{(eq. 1)}
\end{array}
$$

Judgment was exercised in the third step, where we decided *not* to replace $\hat{z}$ by r_z; and in the fourth step, where a subterm was rewritten. Including x in the incrementalization was unnecessary because its value never changes. In Section 4 we will narrow our attention to those state variables that benefit from incrementalization. As in the previous example, the increment is now incorporated and folded in the original loop to obtain the result in Section 1.2.

3.3 More about incorporation

Since we are incrementalizing a loop, incorporating and folding are automatable [14]. Briefly, in the loop

$x := X_0;$
while T **do** $x := b(x);$
output(x)

the increment of the body $b(x)$ introduces r as a trailer variable:

$x := X_0;$
if T **then**
  $(x,r) := \langle b(x), x\rangle;$
  while T **do** $x, r := b'(x,r), x;$
output(x)

We are often able to rewrite the test T, the update $b'(x,r)$, and the *output* combination exclusively in terms of r (denoted T^+, $b^+(r)$ and $output^+$, respectively); hence, x can be eliminated

$r := X_0;$
if T^+ **then**
  $r := b^+(r);$
  while T^+ **do** $r := b^+(r);$
  output$^+(r);$

In practice, we rename r (or its components) to x (or its components). It is also usually possible to fold some or all of r's initialization, as well as some or all of the *output*$^+$ combination back into the loop. In summary, loop incrementalization involves three phases:

1. Move expensive terms out of tests, introducing variables as needed.
2. Solve the incrementalization problem using the loop body as the state mutator.
3. Incorporate, simplify, and fold the solution.

4 Application to sqrt [17]

Figure 1 shows the source and target expressions of sqrt, a *nonrestoring integer square root* algorithm, verified in Nuprl by O'Leary, Leeser, Hickey and Aagaard [17]. In this section we show the details of a formal derivation based on incrementalization. This derivation was also performed in CACHET, as discussed in the next section.

Of course, it should first be established that the specification is correct. O'Leary, et.al prove in Nuprl in that the result is correct, except possibly in the least significant bit [17]. That is, *For any proper input, x,*

$$(\mathsf{sqrt}(x)-1)^2 \le x < (\mathsf{sqrt}(x)+1)^2$$

Since we are optimizing a loop, incrementalization specializes to the case that the function F and state mutator $\oplus$ are the same. In this case, the variable n is unchanged, and the loop index i decrements independently of other variables. Let us therefore focus on the update to m, denoted by M below.

$$F(n,m,i) = \oplus\langle n,m,i\rangle = \langle n, M(n,m,i), i-1\rangle$$

where

$$\begin{aligned} M(n,m,i) = \ & \textbf{let } p = n - m^2 \textbf{ in} \\ & \textbf{if } p > 0 \textbf{ then } m + 2^i \\ & \textbf{else if } p < 0 \textbf{ then } m - 2^i \\ & \textbf{else } m \end{aligned}$$

4.1 Incrementalization

A caching extension of M is

$$\begin{aligned} & \overline{M}(n,m,i) \stackrel{\diamond}{=} \\ & \quad \textbf{let } p = n - m^2 \textbf{ in} \\ & \quad \textbf{let } u = 2^i \textbf{ in} \\ & \quad \textbf{if } p > 0 \textbf{ then} \\ & \quad\quad \langle m + u, p, u, 2mu, u^2 \rangle \\ & \quad \textbf{else if } p < 0 \textbf{ then} \\ & \quad\quad \langle m - u, p, u, 2mu, u^2 \rangle \\ & \quad \textbf{else } \langle m, 0, -, -, - \rangle \end{aligned}$$

The extension includes two *auxiliary* values, $2mu$ and u^2, that do not contribute directly to M's result but do contribute to the computation of $M \circ \oplus$. In order to see this, consider

$$\begin{aligned} & F(\oplus(n,m,i)) \\ & = F(n, M(n,m,i), i-1) \\ & = \langle n, M(n, M(n,m,i), i-1), i-2 \rangle \end{aligned}$$

The subterm $M(n, M(n,m,i))$ expands to

$$\begin{aligned} & \textbf{let } p = n - m^2 \textbf{ in} \\ & \textbf{if } p > 0 \textbf{ then} \\ & \quad \textbf{let } p = n - (m + 2^i)^2 \textbf{ in} \\ & \quad \textbf{if } p > 0 \textbf{ then } (m + 2^i) + 2^{i-1} \\ & \quad \textbf{else if } p < 0 \textbf{ then } (m + 2^i) - 2^{i-1} \\ & \quad \textbf{else } (m + 2^i) \\ & \textbf{else if } p < 0 \textbf{ then } \ldots \end{aligned}$$

From this it can be seen that $u = 2^i$ is a candidate for caching. $M \circ \oplus$ also computes

$$n - (m \pm u)^2 \stackrel{!?}{=} n - m^2 \mp 2mu - u^2 \qquad (2)$$

and so $\overline{M}$ saves $2mu$ and u^2. The partial result m^2 is pruned because it is not used separately. Caching auxiliary information is discussed in greater detail in [11].

We arrive at the cached version of M shown in Figure 5, lower left. The next goal is to incrementalize $\overline{M}$ with respect to $\oplus = \overline{M}$, that is, compute

$$\begin{aligned} (m', p', u', v', w') &= \overline{M}(n, m, i-1) \ \textit{where} \\ (m, p, u, v, w) &= \overline{M}(n, m_0, i) \end{aligned}$$

As discussed at the end of Section 3, we would like an increment involving only cached values. Forming the increment involves two applications of judgment. In $\overline{M}$, w maintains the value u^2 (Fig. 5, lower left); so in $\overline{M}'$

$$w' = (u')^2 = \left(\frac{u}{2}\right)^2 \stackrel{!?}{=} \frac{w}{4} \qquad (3)$$

These derivations take place in context of the tests that guard them. Thus, for example, in the case that both $p > 0$ and $p' > 0$ the fourth component of the result is

$$\begin{aligned} 2m'n' &= 2(m+u)\tfrac{u}{2} \\ &\stackrel{!?}{=} \tfrac{2mu}{2} + \tfrac{2u^2}{2} \\ &= \tfrac{v}{2} + w \end{aligned} \qquad (4)$$

The remainder of $\overline{M}'$ is given in Figure 5.

4.2 Incorporating, folding, and simplifying

Incorporating the incrementalized result in the original program loop opens opportunities to optimize in three ways, each based on dependence analyses already used in incrementalization, and each possibly involving tactical judgment. The goal is to eliminate unneeded terms.

1. *Replace the loop test.* $\overline{M}'$ no longer refers to the loop index, i. If we can remove i from the test (Fig. 1, left), it is no longer needed at all. In $\overline{M}$ u's role is to maintain 2^i and w maintains u^2. Hence,

$$\begin{aligned} i' \geq 0 &\iff i \geq 1 \\ &\iff u \geq 2 \\ &\stackrel{!?}{\iff} u^2 \geq 4 \\ &\iff w \geq 4 \end{aligned} \qquad (5)$$

   Thus, i is unneeded since we can use either u or w.
2. *Minimize maintained information.* On termination the loop test fails, that is,

$$i' = -1 \iff i = 0 \iff u = 1 \stackrel{!?}{\iff} w = 1 \qquad (6)$$

   Furthermore the only value needed is m, the first component of $\overline{M}'$, which depends on the *previous* value of m, u, and p. If $u = 1$ then, since v maintains $2mu$, we can recover this value as

$$\begin{aligned} m = \ & \textbf{if } p > 0 \\ & \quad \textbf{then } \tfrac{v}{2} + 1 \\ & \quad \textbf{else if } p < 0 \\ & \quad\quad \textbf{then } \tfrac{v}{2} - 1 \\ & \quad\quad \textbf{else } \tfrac{v}{2} \end{aligned}$$

   Analyzing the dependencies in $\overline{M}'$ we determine that the only values needed to maintain p and v are components p, v, and w. Thus, if we choose w to compute the loop test in the preceding step, we can also prune u.

$\oplus = M$	*original*	*increment*
original	$M(n,m,i) =$ **let** $p = n - m^2$ **in** **if** $p > 0$ **then** $m + 2^i$ **else if** $p < 0$ **then** $m - 2^i$ **else** m	*not used*
caching pruned	$\overline{M}(n,m,i) \stackrel{\diamond}{=}$ **let** $p = n - m^2$ **in** **let** $u = 2^i$ **in** **if** $p > 0$ **then** $\langle m + u, p, u, 2mu, u^2 \rangle$ **else if** $p < 0$ **then** $\langle m - u, p, u, 2mu, u^2 \rangle$ **else** $\langle m, p, -, -, - \rangle$	$\overline{M}'(m,p,u,v,w) \stackrel{\diamond}{=}$ **if** $p > 0$ **then** **let** $p = p - v - w$ **in** **if** $p > 0$ **then** $\langle m + \frac{u}{2}, p, \frac{u}{2}, \frac{v}{2} + w, \frac{w}{4} \rangle$ **else if** $p < 0$ **then** $\langle m - \frac{u}{2}, p, \frac{u}{2}, \frac{v}{2} + w, \frac{w}{4} \rangle$ **else** $\langle m, p, \frac{u}{2}, \frac{v}{2} + w, \frac{w}{4} \rangle$ **else if** $p < 0$ **then** **let** $p = p + v - w$ **in** **if** $p > 0$ **then** $\langle m + \frac{u}{2}, p, u, \frac{v}{2} + w, \frac{w}{4} \rangle$ **else if** $p < 0$ **then** $\langle m - \frac{u}{2}, p, u, \frac{v}{2} + w, \frac{w}{4} \rangle$ **else** $\langle m, 0, \frac{u}{2}, \frac{v}{2} + w, \frac{w}{4} \rangle$ **else** $\langle m, p, \frac{u}{2}, \frac{v}{2} + w, \frac{w}{4} \rangle$

Fig. 5. Incrementalization of *sqrt*

3. *Fold and initialize.* The body of the loop in sqrt's implementation, Figure 1, incorporates a version of $\overline{M}'$ with values m and u pruned.

4.3 Review of the example

Judgment, in the form of equational reasoning, was involved in all the steps of incrementalization. Caching, (2 and 3) incrementalization (4), incorporation and folding (5 and 6), all entailed theorems depending on arithmetic identities. Induction was not explicit, although it might be argued that folding is an inductive tactic. The entailed dependence analyses are already provided for incrementalization.

5 CACHET: an incrementalization tool

CACHET [9,14] is a program transformation tool developed to explore and demonstrate incrementalization. Figure 6 shows snapshots of CACHET in operation, as it is applied to the sqrt example.

The primary window in CACHET is a syntax-directed program editor; the cursor addresses and operates on subexpressions according to the program grammar. Subwindows may be opened to manipulate subexpressions, display analyses, and so forth. The CACHET commands allowed in a given editing context are displayed as buttons in a second subwindow, as shown in the figure. Subwindows inherit the contingencies of conditional tests, **let** bindings, and the accumulation of new function definitions. Thus, one can symbolically expand and manipulate a subexpression according to those conditions in effect where it occurs.

CACHET has limited rewriting capability. It can be programmed with a collection of identities to apply in simplifying a term. Thus, derivations once established can often be reexecuted automatically under minor changes to the specification. However, it does not have general facilities for algebra nor any built-in deciders for logic, arithmetic, etc.

Figure 6 shows the key steps in a CACHET derivation of sqrt. The series covers a sequence of sixteen CACHET actions, most being operations invoked by clicking on a button, but a couple involving editing operations in the expression window. It should also be mentioned that the derivation involved setting up a design specific file of algebraic identities to steer term rewriting.

(a) The `update` function is the cached extension of the function M of Section 4.
(b) `Update` is incremented with respect to itself to obtain `update1`.
(c) In this step, we have backtracked to add $2m2^i$ and 2^{i^2} to the cache set. These are the auxiliary values

indirectly generated in Section 4 by expanding the original increment (cf. (b)).

(d) Once again, `update` is expanded with respect to its state mutator to begin derivation of the increment. One of the conditional branches is selected for specialization.

(e) The goal is to replace terms in the function body with components of `R` The `CACHE SET` lists values maintained in `R` and its components.

(f) This frame shows an interaction in which the operator has invoked an identity to rewrite $2(m + 2^i)2^{i-1}$ as $2m2^{i-1} + 2 \cdot 2^i \cdot 2^{i-1}$.

(g) With subterms now in appropriate form, CACHET performs replacement from the cache set...

(h) Resulting in the increment.

Enhancement of CACHET to support incorporating, folding, and optimizing the increment is underway.

6 Conclusions and directions

Incrementalization is a powerful technique, applicable to general recursion schemes and, hence, to a broad class of software and hardware specifications. Understanding its full generality is not necessary if its use is restricted to strength reduction; hence, CACHET may be too powerful a tool for refining hardware specifications as they are currently expressed. The same is often said about using theorem provers in hardware verification. As tools and methods for hardware design progress, and as codesign introduces software to the system specification task, the character of specification will change. As it does, the need will arise for higher levels of abstraction and more general reasoning tools.

In evaluating any verification technique, one should distinguish ingenious interactions—those requiring insight, planning, and judgment—from the routine interactions imposed by the proof strategy. One can then work toward automating the logical "boilerplate" and providing shortcuts and visual support for creative intervention. Many proof assistants have a strategy language for this purpose, and one can also use scripting facilities to make arguments reusable.

This study demonstrates that derivational formalism, conducted in in an appropriate context, is an effective verification method. In this study, that context was incrementalization specialized to strength reduction. We contrasted the derivational proof of a nonrestoring square-root computation with its deductive proof in Nuprl. The Nuprl proof of the behavioral specification also follows a strength reduction paradigm, represented by invariant strengthening, as sketched in Section 1.2.

The algorithm implementation in Figure 1 is the same as that of O'Leary, et. al. [17]. There, the verification proof is carried into the architectural level using an embedded hardware description language called *HML*. Some optimizations are performed on the architecture. For instance, elimination of the loop index, i is done in there, rather than in the algorithm. Formal derivation and subsequent refinement of a correct architecture for Figure 1 is straightforward in the DDD algebra. Numerous examples at a similar level of abstraction have been published [6,2,15]. However, one future direction for this work is to compare performing refinements on the behavioral and architectural sides.

CACHET is a research prototype developed, primarily, to investigate and demonstrate incrementalization algebra. We are only beginning to explore its use as an interactive design tool. The `sqrt` case study reveals a number of issues for continued study.

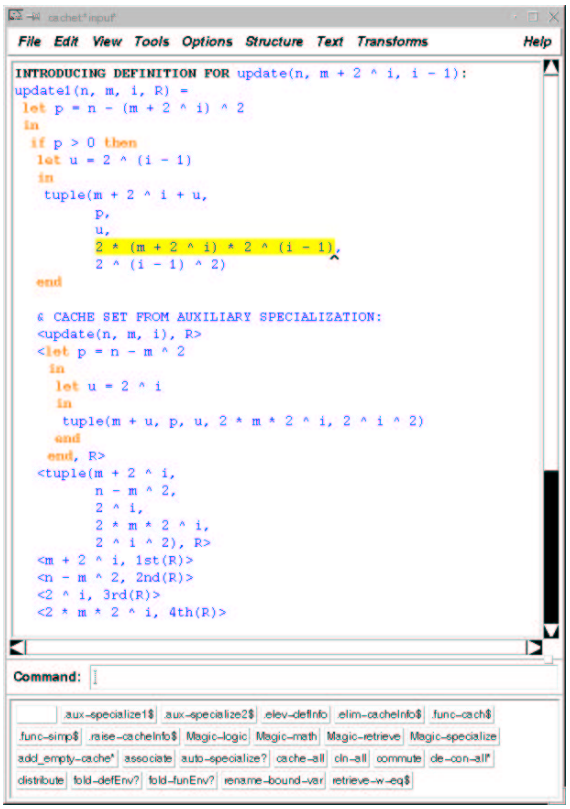
A specialized mode for strength reduction would make the tool easier to use in hardware applications. In software, the "big wins" often come from recursion removal either in control or data. In hardware, there is usually no recursion to remove, and optimizations come from clever reuse of partial values. This difference in focus should be reflected in the tool. For example, in the `sqrt` derivation, we had to backtrack to manually extend the cache set. This step can be automated.

Both CACHET and DDD need greater flexibility in integrating both logical and equational reasoning facilities. Justifications 2 through 6 in Section 4 illustrate points in the derivation where one would like attach deductive reasoning, perhaps even a proof assistant. Conversely, we have found long algebraic derivations to be difficult to execute in sequent style proof assistants. Even more difficult are arguments involving systems, which are best represented by simultaneous recursive definitions.

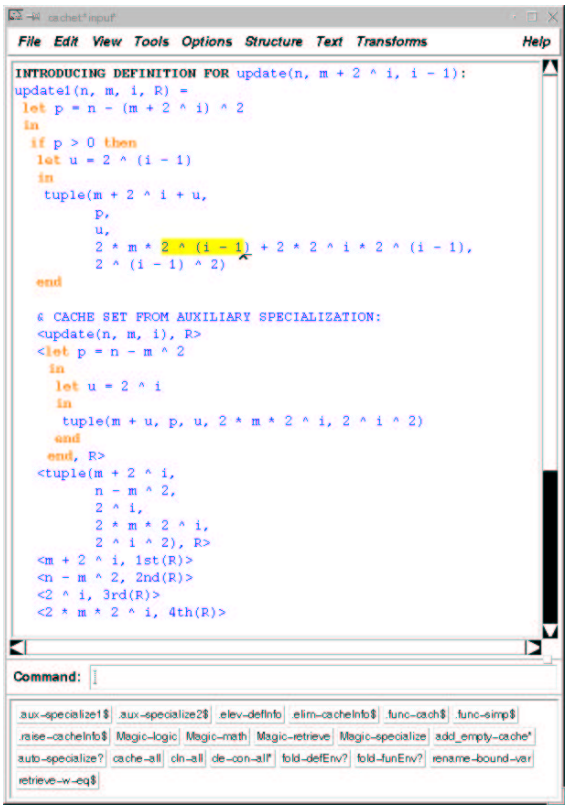
Notes

1. (p. 5) There are some minor departures from conventional **while**-program syntax. We rely on indentation, rather than **begin**–**end** brackets, to depict the scope of compound statements; and we use explicit tuples, $(v_1, v_2) := \langle \mathcal{E}_\infty, \mathcal{E}_\in \rangle$ to express parallel assignment.
2. (p. 4) Liu gives the state mutator $\oplus$ the signature $\oplus\colon W \times Y \to W$, allowing for the introduction of external "inputs" from a set Y. External inputs are not used in any of our examples, so we simplify $\oplus$ for this exposition.

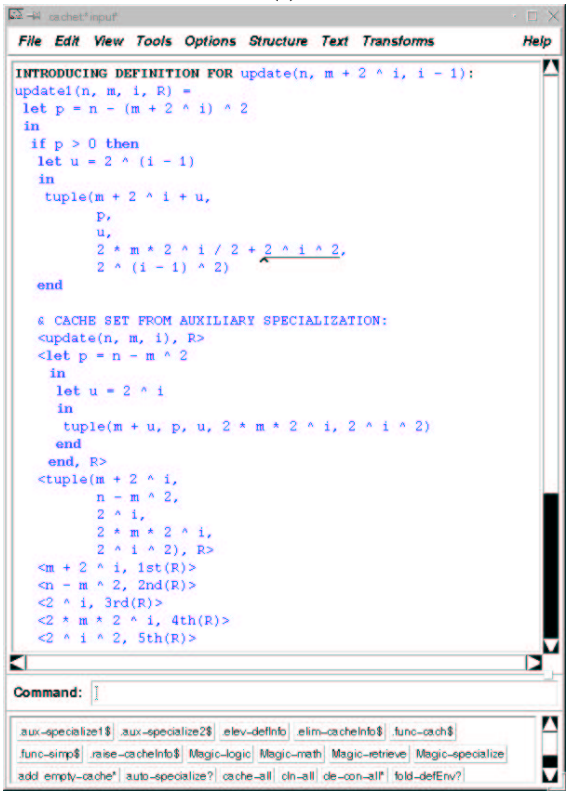
Fig. 6. Key steps in the CACHET derivation of sqrt



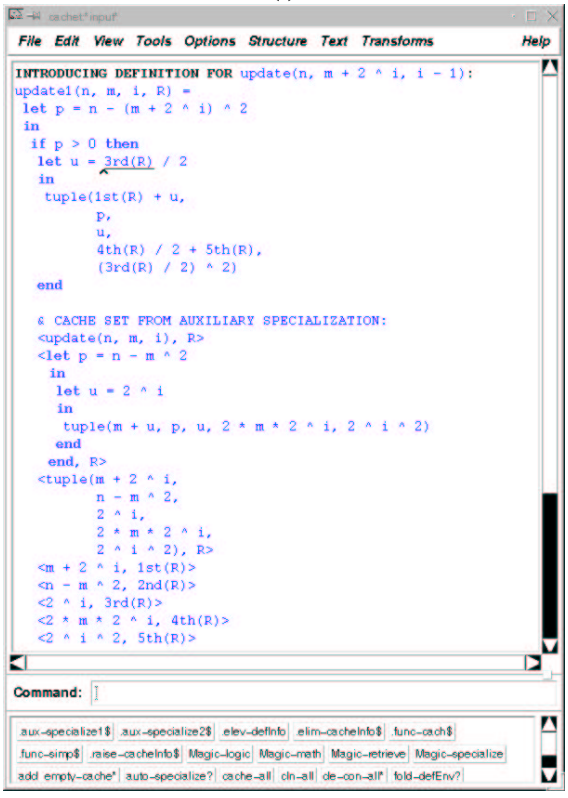
(e)



(f)



(g)



(h)

Figure 6 continued

Acknowledgements. We are indebted to Warren Hunt and John O'Leary for their comments on early versions of this article.

References

1. Bhaskar Bose. *DDD-FM9001: Derivation of a Verified Microprocessor*. PhD thesis, Computer Science Department, Indiana University, USA, December 1994. Technical Report No. 456, 155 pages.
2. Bhaskar Bose and Steven D. Johnson. DDD-FM9001: Derivation of a verified microprocessor. an exercise in integrating verification with formal derivation. In G. Milne and L. Pierre, editors, *Proceedings of IFIP Conference on Correct Hardware Design and Verification Methods*, pages 191–202. Springer, LNCS 683, 1993.
3. Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
4. David A. Greve. Symbolic simulation of the JEM1 microprocessor. In Ganesh Gopalakrishnana and Phillip Windley, editors, *Formal Methods in Computer Aided Design (FMCAD'98)*, volume 1522 of *Lecture Notes in Computer Science*, pages 321–333. Springer, 1998.
5. David Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.
6. Steven D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In *IFIP and ACM/SIGDA International Workshop on Formal Methods in V LSI Design*, 1991. Available as Indiana University Computer Science Department Technical Report No. 323 (rev. 1997).
7. Steven D. Johnson and Paul S. Miner. Integrated reasoning support in system design: design derivation and theorem proving. In Hon F. Li and David K. Probst, editors, *Advances in Hardware Design and Verification*, pages 255–272. Chapman-Hall, 1997. IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'97).
8. Boert B. Jones, Jens U. Skakkebaek, and David L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In Ganesh Gopalakrishnana and Phillip Windley, editors, *Formal Methods in Computer Aided Design (FMCAD'98)*, volume 1522 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 1998. Second International Conference, FMCAD'98.
9. 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*, pages 19–26, Boston, Massachusetts, November 1995. IEEE CS Press, Los Alamitos, Calif.
10. Yanhong A. Liu. Principled strength reduction. In Richard Bird and Lambert Meertens, editors, *Algorithmic Languages and Calculi*, pages 357–381. Chapman & Hall, London, U.K., 1997.
11. Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. Discovering auxiliary information for incremental computation. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 157–170, St. Petersburg Beach, Florida, January 1996. ACM, New York.
12. Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. and Syst.*, 20(2):1–40, March 1998.
13. Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, February 1995.
14. Yanhong Annie Liu. *Incremental Computation: A Semantics-Based Systematic Transformational Approach*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, January 1996.
15. Paul S. Miner. *Hardware Verification using Coinductive Assertions*. PhD thesis, Computer Science Department, Indiana University, USA, 1997. To appear shortly.
16. J Strother Moore. Symbolic simulation: an ACL2 approach. In Ganesh Gopalakrishnana and Phillip Windley, editors, *Formal Methods in Computer Aided Design (FMCAD'98)*, volume 1522 of *Lecture Notes in Computer Science*, pages 334–350. Springer, 1998.
17. John O'Leary, Miriam Leeser, Jason Hickey, and Mark Aagaard. Non-restoring integer square root: A case study in design by principled optimization. In Ramayya Kumar and Thomas Kropf, editors, *Proceedings of the 2nd International Conference on Theorem Provers in Circuit Design: Theory, Practice, and Experience*, volume 901 of *Lecture Notes in Computer Science*, pages 52–71, Bad Herrenalb (Black Forest), Germany, September 1994. Springer-Verlag, Berlin.
18. Phillip Wadler and John Hughes. Projections for strictness analysis. In *3rd International Conference on Functional Programming Languages and Computer Architecture*, pages 385–407, Berlin, 1987. Springer *LNCS* 274.
19. Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27:164–180, 1980.
20. Mitchell Wand. *Induction, Recursion and Programming*. North Holland, 1980.
21. Phillip Windley, Mark Aagard, and Miriam Leeser. Towards a super duper hardware tactic. In Jeffery J. Joyce and Carl Seger, editors, *Higher-Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1993.