Technical Report No. 524

A Systematic Incrementalization Technique and its Application to Hardware Design

by

Steven D. Johnson
Yanhong A. Liu
Yuchen Zhang

June 1999

Computer Science Department
Indiana University
Bloomington, Indiana 47405-4101

A systematic incrementalization technique and its application to hardware design

Steven D. Johnson* and Yanhong A. Liu** and Yuchen Zhang
Computer Science Department
Indiana University

Abstract. A systematic transformation method based on *incrementalization* and value *caching*, generalizes a broad family of program improvement techniques. The technique and an interactive tool supporting it are presented. Though highly structured and automatable, better results are obtained through interaction with an external intelligence, whose main task is to provide insight and proofs involving term equality. This process yields significant performance improvements in many representative program classes, including iterative schemes that characterize Today's hardware specifications. This is illustrated by the derivation of a hardware-efficient nonrestoring square-root algorithm.

Keywords and Phrases: Formal methods, hardware verification, design derivation, formal synthesis, transformational programming, floating point operations.

1 Introduction

The transformation technique described in this paper is familiar, in some form, 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 formally.

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 [16]. That, too, was a tutorial methodological illustration using formalized reasoning in a hardware oriented design context. The purpose of our comparison is to explore how the critical insights needed to justify an implementation are discovered and introduced under deductive and derivational styles of reasoning.

This is not a question of which style is better, but of understanding how intelligent judgements are made and how these activities are reflected in the reasoning tools. The ultimate goal is a reasoning environment supporting a variety of reasoning systems, both automatic and interactive. In order to successfuly reach that goal, we seek better understanding of how human interaction is reflected in reasoning processes. The SQRT example is relatively simple, but it is representative of real designs in signal processing, arithmetic units, microprocessor pipelines, and so on. The essence the verification, the key insights, are of algebraic identities—in this case, laws of arithmetic, but generally equational laws of a given abstract data type. So we are interested how these are discovered and, once discovered, applied in a given reasoning framework.

This paper has two main goals. The first is to introduce the analyses and constructions make up incrementalization. We begin with an small motivating example relating it to *stength 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 the software domain, application of incrementalization has been shown to yield dramatic asymptotic performance improvements (e.g. [11]) by enabling recursion removal.

Loop strength reduction is an important special case. The second goal is to illustrate how incrementalization specializes to the iterations common in hardware specification. In this context an incrementalization tool, like CACHE introduced in Section 3, facilitates the interplay of designer insight with formal manipulation. A full exposition of incrementalization is covered in the several highly technical papers, cited earlier. The examples developed here also appear in those papers. Our main purpose is to present incrementalization as a formal method applied in a restricted specification domain.

* Supported, in part, by the National Science Foundation under grant MIP-9601358.
** Supported, in part, by the National Science Foundation under grant CCR-9711253

1.1 Background

The core approach to *incrementalization* is described by Liu in her dissertation [12, 13]. An incrementalization tool, *CACHE* is the focus of [8]. Subsequently, extensions to the basic approach have addressed with *caching*, or maintaining partial results in auxiliary variables [10, 11]. Caching uses an on-line dependence analysis to prune unneeded accumulators. In [9], Liu outlines the steps of a systematic, semi-automatable incrementalization process, including a presentation of the *sqrt* derivation used here in Section 5.

The articles cited above give extensive reviews of related literature, of which there is a great deal, since incrementalization unifies a large class of program improvement techniques. In particular, incrementalization restricted to iterative (or looping) constructs it is analogous to *strength reduction*. These are the kinds of patterns we expect to see in hardware specification, so the question arises whether incrementalization specialized to strength reduction is useful.

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 theorem prover, in this instance *PVS*. It is demonstrated in [2, 1] that such a heterogeneous framework reduces the effort of verifying a microprocessor implementation. In [6, 14], a more tightly coupled relationship between a derivational and deductive formalisms is explored.

Both DDD and CACHE are design derivation tools, operating on similar, but not, identical formal languages. In the main, they apply to different aspects of design, so it is reasonable to consider integrating them in a single framework. CACHE applies to a control oriented expression of behavior, while DDD's principle purpose is to build and manipulate architecture oriented expressions.

In the past few years, there has increasing attention to term-level reasoning in hardware verification. Validity checking with uninterpreted function symbols (e.g. [7]), 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 the essence of hardware verification proofs lies in equational reasoning at the level of terms but in the context of systems. Moore's description of a *symbolic spreadsheet* [15] reflects this insight. Greve's use of symbolic simulation in the JEM1 microprocessor verification, and other similar case studies are exploring interactive verification methods centering on presenting algebraic identities to the engineer [4].

In 1993, Windley, Leeser, and Aagard pointed out that numerous hardware verifications have been found to follow a common proof plan [19]. Incrementalization might be seen as a "super duper" derivation tactic, but one that is applicable to a very broad range of specification patterns.

1.2 Strength reduction

Incrementalization generalizes a basic programming technique found in virtually all treatments of programming, however formal. The motivating illustration below comes an undergraduate textbook written in 1978 [18], which credits Dijkstra for the phrase "strength reduction" [3].

We want an algorithm to compute the integer square root of an input x; that is, an S such that $\{0 \leq x\}\ S\ \{z^2 \leq x < (z+1)^2\}$.** The *and* in the postcondition suggests a loop with one conjunct providing the test and the second an invariant [5]:

$$
\begin{array}{l}
z := 0; \\
\textbf{while } x \geq (z+1)^2 \textbf{ do } \{z^2 \leq x\}\ z := z+1
\end{array}
$$

To get rid of the expensive term $(z+1)^2$, we can introduce an auxiliary variable u to hold this value. The invariant becomes $\{z^2 \leq x \textit{ and } u = (z+1)^2\}$ 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)

$$
z' = z + 1 \qquad \text{and} \qquad
\begin{array}{ll}
u' = (z'+1)^2 & \\
\quad = z'^2 + 2z' + 1 & (?) \\
\quad = (z+1)^2 + 2z' + 1 & \\
\quad = u + 2z' + 1 & (!)
\end{array}
$$

** We use brackets $\{\cdots\}$ to express partial correctness here.

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

$$z, u := 0, 1;$$
$$\textbf{while } x \geq u \textbf{ do } z, u := z + 1, u + 2(z + 1) + 1$$

There are four discussion points.

First, u exploits the algebraic identity, $(x+1)^2 = x^2 + 2x + 1$, at the third step in the derivation of u'. In general, we cannot expect such insight be fully automated. From here on, we refer to the application of algebraic laws an *exercise of judgment*, presumably by an ingenious agent. We indicate points of judgements with the symbol ' $\stackrel{!?}{=}$ '. However, even if *no* such interventions take place, some programs are improved by saving 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.

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 the term $2z$ 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 *multipy-by-two* to *add-two*, provided we can see to apply the distributive law. Obviously, this is not an improvement for hardware, and whether it is or isn't for software depends on the compiler. Thus, judgment may also be needed in the tactical application of incrementalization.

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 5.

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. On the other hand one measure of a good method is that it can be applied to a range of special circumstances.

2 Systematic incrementalization

In this section we survey the general approach to incrementalization, and we revisit applications to strength reduction in Section 4.

Formally, 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. Most of our examples involve arithmetic operations with the usual laws of algebra. Generally, the structure is an abstract data type.

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, our examples include an alternative **while**-program notation, in the style of Gries [5], for those who are more comfortable with that form of expression.

The incrementalization method is actually an interplay between two kinds of function extension, *incrementalization* and *caching*, as indicated in Figure 1. Let $F\colon W \to V$ and $oplus\colon (W \times Y) \to W$. F plays the role of the specification we are transforming and $\oplus$ is some *state mutator*, or arbitrary combination of elementary operations applied to F's argument.

$\oplus: W \times Y \to W$	*original*	*incrementalized*
original	$F: W \to V$	$F': W \times Y \times V \to V$ $F'(w, y, F(w)) = F(w \oplus y)$
caching	$\overline{F}: W \to V^n$ $F(w) = v_1$ **where** $\langle v_1, v_2, \ldots \rangle = \overline{F}(w)$	$\overline{F}': W \times Y \times V^n \to V^n$ $\overline{F}'(w, y, \overline{F}(w)) = \overline{F}(w \oplus y)$

Fig. 1. Components of incrementalization. *These are not defining equations, but identities relating cached, incrementalized, and cached-incrementalized variants of F.*

The *incrementalization of F with respect to $\oplus$* is a function that computes $F(w \oplus y)$ given the value of $F(w)$. That is, $F': (W \times Y \times V) \to V$ has the property that

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

The idea is this: given a specification for F by which computing $F(w \oplus y)$ involves a recursive call to $F(w)$, we want to consider how $F(w)$ is used in calculating the final result. Suppose that F builds a data structure; then incrementalization involves analyzing how parts of the object $F(w)$ are reused in creating the object $F(w \oplus y)$. This dependence analysis may require unrolling chains of recursive calls far enough to identify sufficient patterns of access, in a manner similar to strictness analysis.

NOTE: Only Example 2 makes use of values introduced from Y. In the other examples, we write $\oplus(w)$, omitting mention of y.

Caching extends a function to return auxiliary results. $F: W \to V$ is extended to $\overline{F}: W \to V^k$, so that $\overline{F}(w) = \langle F(w), v_2, \ldots, v_k \rangle$. The auxiliary values will be those partial results determined by dependence analysis to be useful across recursive calls. That is, in incrementalization, the cached values, v_i will be components of, formally, subterms of $v_1 = F(w)$.

Thus, what we are really after is $\overline{F}'$, the incrementalization of the caching extension of F, or, if you wish, the caching extension of the incrementalized call to F. It is important to understand that this transformation is being applied to an applied occurrence of F, one of possibly many points where F is called. It then remains to incorporate $\overline{F}'$ in the original specification of F, as we will see in the examples.

2.1 Example 1 – Application to recursive specifications

We begin with the "*fibonnaci*" scheme, below left. and incrementalized with respect to the term $\oplus(x) \stackrel{\diamond}{=} x+1$. The choice of $\oplus$ reflects the fact that incrementalizing F means computing $F(x+1)$ given $F(x)$. Looking at the original defining scheme, another candidate for $\oplus(x)$ might be $x+2$. In other words, the choice of $\oplus$ is usually straightforward.

$$\begin{array}{lll} F(x) \stackrel{\diamond}{=} & \textbf{if } x \leq 1 & \\ & \textbf{then } 1 & \\ & \textbf{else } F(x-1) + F(x-2) & \end{array} \qquad \begin{array}{lll} F'(x, r) \stackrel{\diamond}{=} & \textbf{if } & x \leq 0 \textbf{ then } 1 \\ & \textbf{else } & \textbf{if } x = 1 \textbf{then } 2 \\ & \textbf{else } & r + F(x-1) \end{array}$$

Incrementalization with respect to $\oplus$, yields F', on the right above, such that $F'(x, F(x)) = F(x+1)$ However, this function doesn't get us very far; the result is still a quadratic recursion pattern. A *cache-and-prune* analysis accumulates not only $F(x)$ but also all relevant intermediate values Assuming addition is "cheap," there are two intermediate values, $F(x-1)$ and $F(x-2)$. A first caching approximation, $\overline{F_0}$, takes the form on the left below with the incrementalized version on the right. Notice that the caching parameter,

r is now a nested data structure. Recall (Fig. 1) that if $r = \overline{F}(x)$ then $\overline{F}'(x, r) = \overline{F}(x+1) = \langle F(x+1), \ldots \rangle$

$$\begin{array}{l} \overline{F_0}(x) \stackrel{\diamond}{=} \\ \quad \textbf{if } x \leq 1 \textbf{ then } \langle 1, \text{-}, \text{-} \rangle \\ \quad \textbf{else } \textbf{let } u = (u_1, u_2, u_3) = \overline{F_0}(x-1) \textbf{ in} \\ \qquad \textbf{let } v = (v_1, v_2, v_3) = \overline{F_0}(x-2) \textbf{ in} \\ \qquad \langle u_1 + v_1, u, v \rangle \end{array}$$

$$\begin{array}{l} \overline{F_0}'(x, r) \stackrel{\diamond}{=} \\ \quad \textbf{if } x \leq 0 \textbf{ then } \langle 1, \text{-}, \text{-} \rangle \\ \quad \textbf{else if } x = 1 \textbf{ then } \langle 2, \langle 1, \text{-}, \text{-} \rangle, \langle 1, \text{-}, \text{-} \rangle \rangle \\ \quad \textbf{else } \textbf{let } (r_1, r_2, r_3) = r \textbf{ in} \\ \qquad \textbf{let } (r_{21}, r_{22}, r_{23}) = r_2 \textbf{ in} \\ \qquad \langle r_1 + r_{21}, r, r_2 \rangle \end{array}$$

Now consider the result, $\langle r_1 + r_{21}, r, r_2 \rangle$. You may be able to see that the value "of interest," $r_1 + r_{21}$ does not depend on subcomponents r_{22}, r_{23}, or r_3. An automatic, on-line analysis of transitive dependence [11] confirms that these elements do not need to be maintained and can be pruned. We get $\overline{F_1}$ on the left, incrementalized with respect to $x+1$ on the right

$$\begin{array}{l} \overline{F_1}(x) \stackrel{\diamond}{=} \\ \quad \textbf{if } x \leq 1 \textbf{ then } \langle 1, \text{-} \rangle \\ \quad \textbf{else} \\ \qquad \textbf{let } (u_1, u_2, u_3) = \overline{F_0}(x-1) \textbf{ in} \\ \qquad \textbf{let } (v_1, v_2, v_3) = \overline{F_0}(x-2) \textbf{ in} \\ \qquad \textbf{in} \\ \qquad \langle u_1 + v_1, \langle u_1 \text{-} \rangle, \text{-} \rangle \end{array}$$

$$\begin{array}{l} \overline{F_1}'(x, r) \stackrel{\diamond}{=} \\ \quad \textbf{if } x \leq 0 \textbf{ then } \langle 1, \text{-} \rangle \\ \quad \textbf{else if } x = 1 \textbf{ then } \langle 2, \langle 1, \text{-} \rangle \rangle \\ \quad \textbf{else } \textbf{let } (r_1, r_2) = r \textbf{ in} \\ \qquad \textbf{let } (r_{21}, r_{22}) = r_2 \textbf{ in} \\ \qquad \langle r_1 + r_{21}, \langle r_1, \text{-} \rangle \rangle \end{array}$$

The structure r is now linear and not tree-like, a positive development. Incorporating these refinements into the original scheme, and doing some elementary restructuring, we obtain at the definition

$$\begin{array}{l} \overline{F}(x+1) \stackrel{\diamond}{=} \textbf{if } x \leq 0 \textbf{ then } \langle 1, \text{-} \rangle \\ \qquad\qquad\qquad\quad \textbf{else if } x = 1 \textbf{ then } \langle 2, 1 \rangle \\ \qquad\qquad\qquad\quad \textbf{else } \textbf{let } (r_1, r_2) = \overline{F}(x) \textbf{ in } \langle r_1 + r_2, r_1 \rangle \end{array}$$

Replace $x+1$ by z and simplify to recover the original form of the conditional. The important outcome is that the result is a linear recursion, derived, not proven inductively, although it certainly could be. Verification is subsumed by the pruning analysis. In fact, we haven't yet used the algebraic properties of addition. It requires associativity to obtain the iterative version of *fibonnaci*. This, too, is derivable, using techniques originating with Wand [17].

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

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

2.2 Example 2 – Caching further extended

Having come this far, let us briefly illustrate a further extension of caching to address auxiliary function calls [10]. The example also serves to demonstrate the more typical use of incrementalization to refine data structures. In the system below, *CMP* compares the sum of odd-position elements and the product of even-position elements in a list of numbers.

$$CMP(x) \stackrel{\diamond}{=} \Sigma(Os(x)) \leq \Pi(Es(x))$$

$$\begin{array}{ll} Os(x) \stackrel{\diamond}{=} \textbf{if } null(x) \textbf{ then } nil & \Sigma(x) \stackrel{\diamond}{=} \textbf{if } null(x) \textbf{ then } 0 \\ \qquad\quad \textbf{else } cons(car(x), Es(cdr(x))) & \qquad\quad \textbf{else } car(x) + \Sigma(cdr(x)) \\ \\ Es(x) \stackrel{\diamond}{=} \textbf{if } null(x) \textbf{ then } nil & \Pi(x) \stackrel{\diamond}{=} \textbf{if } null(x) \textbf{ then } 1 \\ \qquad\quad \textbf{else } Os(cdr(x)) & \qquad\quad \textbf{else } car(x) \times \Pi(cdr(x)) \end{array}$$

Suppose we have reason to incrementalize *CMP* with respect to

$$y \oplus x \stackrel{\diamond}{=} cons(y, x)$$

Here we see the introduction of the free variable y in the state modifier. Even if $\Sigma(Os(x))$ and $\Pi(Es(x))$ are cached, $CMP(cons(y, x))$ remains expensive because even positions are now odd and vice versa. Hence, Σ and π have to be recomputed. Caching the subterms in *CMP*'s defining expression is not sufficient to improve this program significantly; we must also capture partial sums and products. The extended caching incrementalization in [10] results in the following:

$$\begin{array}{l}\overline{CMP}(x) \stackrel{\diamond}{=} \textbf{let } v_1 = Os(x) \textbf{ in} \\ \qquad\qquad\quad \textbf{let } v_2 = Es(x) \textbf{ in} \\ \qquad\qquad\quad \textbf{let } u_1 = \Sigma(v_1) \textbf{ in} \\ \qquad\qquad\quad \textbf{let } u_2 = \Pi(v_2) \textbf{ in} \\ \qquad\qquad\quad \langle u_1 \leq u_2, u_1, u_2, v_1, v_2\rangle\end{array}$$

$$\begin{array}{l}\overline{CMP}'(y, (r_1, r_2, r_3, r_4, r_5)) \stackrel{\diamond}{=} \\ \quad \langle (y + r_4) \leq r_5,\ y + r_4,\ r_5,\ r_2,\ y \times r_3\rangle\end{array}$$

In $\overline{CMP}'$ partial sums and products are cached, to be used when a number y is added to the list. Hence, the incremental cost of computing $CMP(y \oplus x)$ goes from linear time to constant time.

3 Incrementalization specialized to strength reduction

Let us return to the the naive *sqrt* from Section 1.2 We present the derivation in both functional and imperative syntax. For clarity, we move the computation of $(z+1)^2$ out of the test and into the body of the loop.

$$\begin{array}{l}sqrt(x) \stackrel{\diamond}{=} S(x, 0, 1) \\ \\ S(x, z, u) \stackrel{\diamond}{=} \\ \quad \textbf{if } x < u \textbf{ then } z \\ \quad \textbf{else } S(x, z+1, ((z+1)+1)^2)\end{array}$$

$$\begin{array}{l}x, z, u := input, 0, 1; \\ \textbf{while } x \geq u \textbf{ do} \\ \quad z, u := z+1, ((z+1)+1)^2; \\ output := z\end{array}$$

There is only one occurrence of S to consider, but what is the $\oplus$ with respect to which S should be incrementalized? Let us (not really arbitrarily) take it to be the update function that S performs,

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

No free values (*y*s) are used, so we have specialized from *oplus*: $(W \times Y) \to W$ to *oplus*: $(W \to W$, where W in this case consists of numeric triples. According to Figure 1, we want to construct S' such that $S'(w, S(w)) = S(\oplus(w))$, but *oplus* was selected such that, in the looping branch, $S(\oplus(w)) = \oplus(\oplus(w))$. In other words, the choice of $\oplus$ amounts to loop unrolling.

Let

$$(x', z', u') \text{ denote } \oplus(x, z, u) = \langle x, z+1, ((z+1)+1)^2\rangle$$

and let

$$(r_x, r_z, r_u) \text{ denote } S(x, z, u) = \langle x, z+1, ((z+1)+1)^2\rangle$$

We can derive† (*Note:* x', z', and u' are not playing the same role here as in the introductory example.)

† The derivation is clearer if we replace occurrences of "$z+1$" by a fresh variable; but we think introducing metavariables at this point would cloud the explanation of incrementalization.

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

The final value is in terms of r. We used judgment in the fourth step applying an algebraic identity, *but also* in the third step, deferring the replacement of one occurrence of $z{+}1$ by r_z.

The incrementalized loop body is now incorporated in the original program. In essence, we perform a two-way "fold". Introducing S' to the system, we get

$$
\begin{array}{l}
sqrt(x) = \\
\textbf{let } (x,z,u) = \langle x, 0, 1 \rangle \\
\textbf{in} \\
\quad \textbf{if } x < u \textbf{ then } z \\
\quad \textbf{else } S'(\langle x, z+1, (z+1)^2 \rangle, \\
\qquad\qquad \langle x, z, u \rangle) \\
\textbf{where} \\
S'((x,z,u),(r_x,r_z,r_u)) = \\
\quad \textbf{if } x < u \textbf{ then } z \\
\quad \textbf{else } S'(\langle r_x, r_z, r_u \rangle, \\
\qquad\qquad \langle r_x, r_z+1, r_u + 2r_z + 1 \rangle)
\end{array}
\qquad
\begin{array}{l}
x,z,u := input, 0, 1; \\
\textbf{if } x \geq u \textbf{ then } z \\
\textbf{else} \\
\quad (x,z,u),(r_x,r_z,r_u) := \\
\qquad \langle x, z+1, ((z+1)+1)^2 \rangle, \langle x, z, u \rangle \\
\quad \textbf{while } x \geq u \textbf{do} \\
\qquad (x,z,u),(r_x,r_z,r_u) := \\
\qquad\quad \langle r_x, r_z+1, r_u + 2r_z + 1 \rangle, \langle x, z, u \rangle; \\
output := z;
\end{array}
$$

Our goal is to get rid of $w = (x,z,u)$. Since $r = (r_x, r_z, r_u)$ is just a trailer variable for w, if we initialized it properly, we could, first, replace the test on w by a test on r, and second, eliminate w entirely by replacing it by r; and then third, rename r to w. We arrive at the anticipated result:

$$
\begin{array}{l}
sqrt(x) \stackrel{\diamond}{=} S(x,0,1) \\
\\
S(x,z,u) \stackrel{\diamond}{=} \\
\quad \textbf{if } x < u \textbf{ then } z \\
\quad \textbf{else } S(x, z+1, 2u + 2(z+1) + 1)
\end{array}
\qquad
\begin{array}{l}
x,z,u := input, 0, 1; \\
\textbf{while } x \geq u \textbf{ do} \\
\quad x,z,u := x, z+1, u + 2(z+1) + 1 : \\
output := z;
\end{array}
$$

Because we knew at the outset that we were incrementalizing a loop, this final collapse is automatic, as long as we can express the next value of r in terms of r. We shall see later that there are opportunities at this point to optimize both the start-up and shut-down phases of the loop. It is also worth noting that including x in the incrementalization is unnecessary. In Section 5 we will narrow our attention to those state variables that benefit from incrementalization.

Once again, the judgment was exercised at just two points in the derivation. Otherwise, we followed these steps:

1. Move expensive terms out of tests, introducing variables as needed.

2. Solve the incrementalization problem. Pick a function call, $F(e_1 \cdots e_k)$ and an state modifier, $\oplus$, to incrementalize. In the iterative case, $\oplus = F = \langle e_1 \cdots e_k \rangle$.

3. Incorporate the solution.

4. Simplify and fold. In Section 5 we will see that this involves three additional substeps.

4 CACHET: an incrementalization tool

CACHET [8,13] is a program transformation tool supporting incrementalization. Figures 3 and 7 are snapshots of CACHET in operation.

The primary window in CACHET is a syntax-directed program editor. The cursor addresses and operates on subexpressions, according to the program grammar, and not characters. Subwindows are used to investigate subprograms and display analyses. For reasoning support, these subwindows inherit the contingencies determined by conditional tests, new **let** bindings, and the accumulation of new function definitions. There is substantial automated support for incrementalization, but, the operator must be familiar with the incrementalization steps to take advantage of the automation.

Figure 3 illustrates an incrementalization of a simple sorting algorithm. The main window displays the system under design, and defines the state modifier and transformation context. The middle window shows a process of expanding and symbolically evaluating the expression under the cursor in the primary window, and similarly in the foremost window. The lower region of the foremost window shows an accounting of cached subexpressions and their identifiers. The buttons are transformations and operations that are meaningful for the selected subexpression. To these may be added term rewrites that are specific to the ground type. Figures 7(a–c) show CACHET applied to the next example, and are discussed in the conclusion.

5 Application to *sqrt* [16]

Figure 7 contains snapshots of the derivation of the *nonrestoring integer square root* implementation, specified by O'Leary, Leeser, Hickey and Aagaard [16]. A more detailed presentation of this derivation can be found in [13]. We show highlights of the derivation in an imperative notation. The initial algorithm (left) and derived implementation (right) are shown together in Figure 2.

$n, i, m := input, (l-2), 2^{l-1};$	$p, v, w := input, 0, 2^{2(l-1)};$
while $i \geq 0$ **do**	**while** $(w \geq 1)$ **do**
$p := n - m^2;$	**if** $p > 0$ **then**
if $p > 0$ **then**	$p, v, w := p - v - w; \frac{v}{2} + w, \frac{w}{4}$
$m := m + 2^i$	**else if** $p < 0$ **then**
else if $p < 0$ **then**	$p, v, w := p + v - w, \frac{v}{2} - w, \frac{w}{4}$
$m := m - 2^i;$	**else**
$i := i - 1;$	$v, w := \frac{v}{2}, \frac{w}{4};$
$output := m$	$output := v$

Fig. 2. Specification and implementation of nonrestoring *sqrt*

Of course, it should first be established that the initial algorithm is correct. O'Leary, et.al prove in Nuprl in that the result is correct, except possibly in the least significant bit [16]:

$$\text{For any proper input, } x, (sqrt(x) - 1)^2 \leq i < (sqrt(x) + 1)^2$$

The goal of the derivation is to refine this specification with hardware in mind. The derivation in CACHET requires a familiarity with the incrementalization strategy and a just a little special knowledge of arithmetic. We will point out where judgment is exercised. The example follows the steps of incrementalization outlined in Section 3.

5.1 Incrementalization

We are optimizing a loop, so incrementalization specializes to the case that the function F and state mutator $\oplus$ (Fig. 1) are the same. In this case, the variable n is unchanged, and the loop index i decrements independently 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

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

A caching extension of M is

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

Let (m,p,u,v,w) denote $\overline{M}(n,m,i)$. Incrementalizing $\overline{M}$ under $\oplus$ yields

$$\begin{aligned} \overline{M}'(m,p,u,v,w) \stackrel{\diamond}{=} \quad & \textbf{if } p > 0 \textbf{ then } \ \textbf{let } p = p - v - w \textbf{ in} \\ & \quad \textbf{if } p > 0 \textbf{ then} \quad \langle m + \tfrac{u}{2}, p, \tfrac{u}{2}, \tfrac{v}{2} + w, \tfrac{w}{4}\rangle \\ & \quad \textbf{else } \ \textbf{if } p < 0 \textbf{ then} \quad \langle m - \tfrac{u}{2}, p, \tfrac{u}{2}, \tfrac{v}{2} + w, \tfrac{w}{4}\rangle \\ & \quad \textbf{else } \ \langle m, 0, \tfrac{u}{2}, \tfrac{v}{2} + w, \tfrac{w}{4}\rangle \\ & \textbf{else } \ \textbf{if } p < 0 \textbf{ then } \ \textbf{let } p = p + v - w \textbf{ in} \\ & \quad \textbf{if } p > 0 \textbf{ then} \quad \langle m + \tfrac{u}{2}, p, u, \tfrac{v}{2} + w, \tfrac{w}{4}\rangle \\ & \quad \textbf{else } \ \textbf{if } p < 0 \textbf{ then} \quad \langle m - \tfrac{u}{2}, p, u, \tfrac{v}{2} + w, \tfrac{w}{4}\rangle \\ & \quad \textbf{else} \quad \langle m, 0, \tfrac{u}{2}, \tfrac{v}{2} + w, \tfrac{w}{4}\rangle \\ & \textbf{else} \quad \langle m, 0, \tfrac{u}{2}, \tfrac{v}{2} + w, \tfrac{w}{4}\rangle \end{aligned}$$

Following Section 3, the initial formal parameter for $\overline{M}'$ is

$$(m,p,u,v,w,(r_m,r_p,r_u,r_v,r_w))$$

but this can be trimmed. Incrementalization results in an update involving only cached values; so we can eliminate m, p, u, v, and w, then rename $((r_m,r_p,r_u,r_v,r_w))$ for legibility. Judgment is exercised at three points in this step.

M computes the intermediate result $n - m^2$ and $\oplus$ updates m to $m \pm u$. Thus, $M \circ \oplus$ computes

$$n - (m \pm u)^2 \stackrel{!?}{=} n - m^2 \mp 2mu - u^2 \tag{1}$$

and so $\overline{M}$ caches $2mu$ and u^2. The partial result m^2 is pruned because it is not used separately. In $\overline{M}$, w maintains the value u^2 and so in $\overline{M}'$

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

In $\overline{M}'$ the 5-tuple returned is simplified according to the context, that is, contingencies determined by the tests. For example, in the first branch, with both $p > 0$ and $p > 0$ the fourth component is

$$2m'n' = 2(m+u)\frac{u}{2} \stackrel{!?}{=} \frac{2mu}{2} + \frac{2u^2}{2} = \frac{v}{2} + w \tag{3}$$

5.2 Incorporating, folding, and simplifying

Incorporating the incrementalized result in the original program 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 elimination of 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. 2, left), it is no longer needed at all. In $\overline{M}$ u's original role was to maintain 2^i, so
$$i' \geq 0 \iff i \geq 1 \iff u \geq 2 \iff u^2 \geq 4 \iff w \geq 4 \tag{4}$$
Thus, i is unneeded since we can use either u or w.
2. *Minimize maintained information.* On termination the loop test fails, that is,
$$i' = -1 \iff i = 1 \iff u = 1 \tag{5}$$
Furthermore the only value needed is m, the first component of $\overline{M}'$, which depends on the *previous* value of m, u, and p. Since $u = 1$ we can recover this value as
$$\textbf{if } p > 0 \textbf{ then } \frac{v}{2} + 1 \textbf{ else if } p < 0 \textbf{ then } \frac{v}{2} = 1 \textbf{ else } \frac{v}{2}$$
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 eliminate u.
3. *Fold and initialize.* The body of the loop in *sqrt*'s implementation, Figure 2, incorporates a version of $\overline{M}'$ with values m and u pruned.

5.3 Review of the example

Judgment, in the form of equational reasoning, was involved in all the steps of incrementalization. Caching, (1 and 2) incrementalization (3), incorporation and folding (4 and 5), 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.

6 Conclusions and directions

Understanding incrementalization in its full generality is substantial undertaking, at least at our current stage of understanding. A more specialized tool for strength reduction might be appropriate to practice, especially in hardware implementation. On the other hand, understanding the more general method may be worth the investment in applications where hardware/software decomposition is involved. It is similar to the issue of whether to use a general purpose theorem prover for special purposes.

Of course, CACHET supports a class of program optimizations, not a complete calculus for program refinement, so the analogy to a theorem prover can only be taken so far. The main point of this study is that the creative input to *sqrt*'s implementation verification boil down to a few algebraic laws, tactically applied. This is the case whether the proof is deductive or derivational.

Figure 7 shows CACHET applied to the *sqrt* example, beginning with the initial representation of the *update* function. In frame (b) we are in the process of developing the cached auxiliary values. In frame (c) the cached partial values are being incorporated in the loop. It is at this stage where, for example, the distributive law for multiplication is invoked to in order to exploit these values. The CACHET tool was developed with a view toward applications to generally recursive software specifications. A specialization to loop forms would suggest some modifications to the user interface, but these appear to be quite superficial. There are other hardware oriented tactics to explore. For example, pipelining is a kind of incrementalization but we do not know whether CACHET can adapt to it.

The implementation in Figure 2 is the same as that of O'Leary, et. al. [16]. In that study, elimination of the loop index is done in the architectural (structural) description, rather than the behavioral one. Formal derivation and subsequent refinement of a correct architecture for Figure 2 is straightforward in the DDD algebra. Whether a given design optimization is better done earlier, in CACHET, or later, in DDD, is a matter for further research.

7 Acknowledgments

We are indebted to Warren Hunt and John O'Leary for their comments on this developing paper.

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 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).
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, February 1995.
13. Yanhong Annie Liu. *Incremental Computation: A Semantics-Based Systematic Transformational Approach.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, January 1996.
14. Paul S. Miner. *Hardware Verification using Coinductive Assertions.* PhD thesis, Computer Science Department, Indiana University, USA, 1997. To appear shortly.
15. 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.
16. 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.
17. Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27:164–180, 1980.
18. Mitchell Wand. *Induction, Recursion and Programming.* North Holland, 1980.
19. 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.

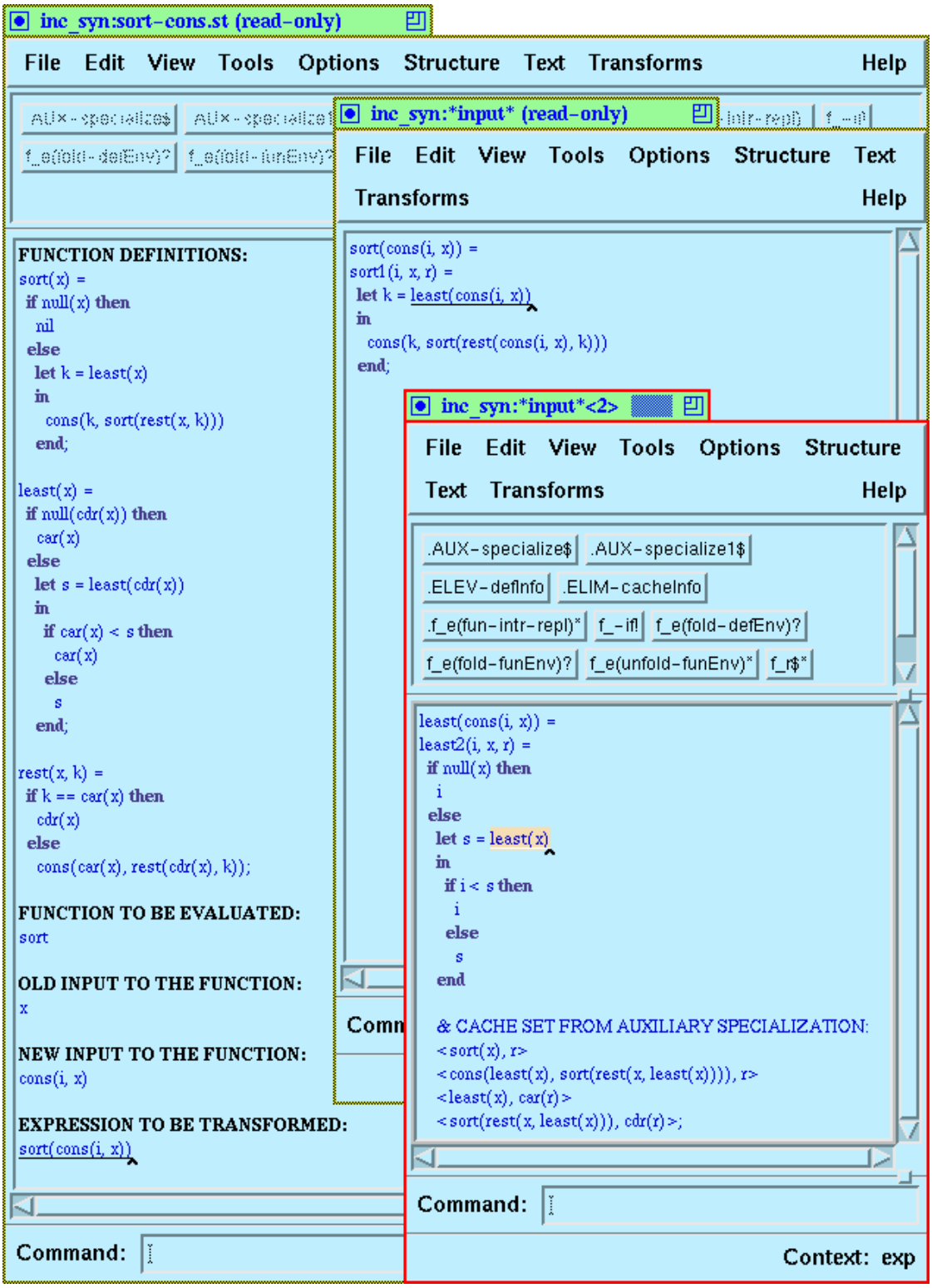
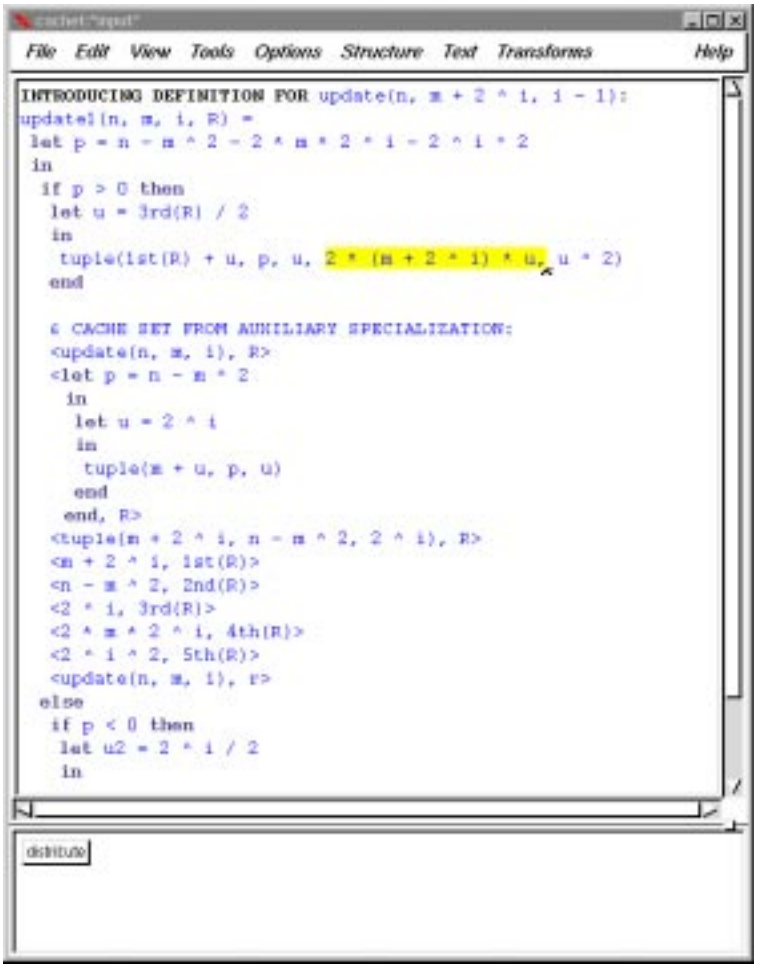
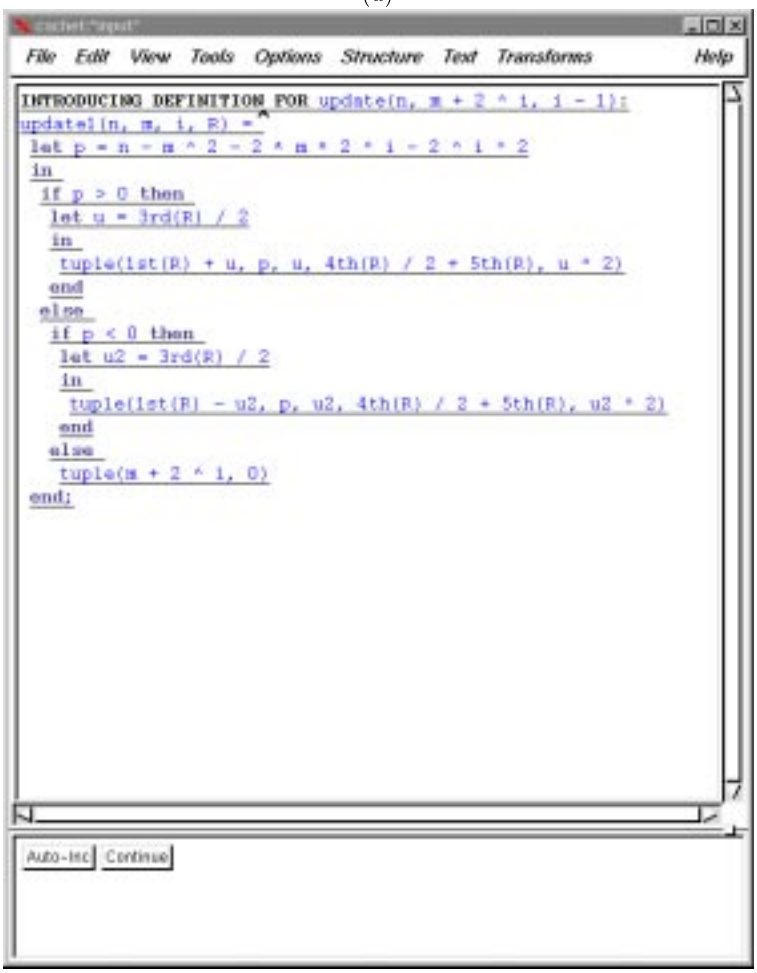


Fig. 3. First view of the CACHET tool.

(a)



(b)



(c)

Fig. 4. Snapshots of the SQRT derivation in CACHET.
Frame (a) shows the initial definition of *sqrt*'s *update* state modifier. In Frame (2), cached values have been determined. In Frame (3) these cached values are being incorporated in the incrementalized computation.