Rewriting for Symbolic Execution of State Machine Models

J Strother Moore*

Department of Computer Sciences, University of Texas at Austin,
Taylor Hall 2.124, Austin, Texas 78712
`moore@cs.utexas.edu` telephone: 512 471 9568
WWW home page: `http://www.cs.utexas.edu/users/moore`

Abstract. We describe an algorithm for simplifying a class of symbolic expressions that arises in the symbolic execution of formal state machine models. These expressions are compositions of state access and change functions and if-then-else expressions, laced together with local variable bindings (e.g., `lambda` applications). The algorithm may be used in a stand-alone way, but is designed to be part of a larger system employing a mix of other strategies. The algorithm generalizes to a rewriting algorithm that can be characterized as outside-in or lazy, with respect both to variable instantiation and equality replacement. The algorithm exploits memoization or caching.

Keywords: hardware modeling, verification, microprocessor simulation, theorem proving, pipelined machine

1 Relevance to Processor Modeling

A common application of such mechanized theorem provers as ACL2 [13], HOL [8] and PVS [17] is the modeling and analysis of microprocessors and other state machines [3, 6, 11, 12, 14–16, 18, 9].

The ACL2 theorem prover [13, 12] is particularly suited to processor modeling because it supports an efficient functional programming language based on Common Lisp [19]. Hence, operational models formalized in ACL2 can be executed as processor simulators. This is not a speculative assertion. Rockwell Collins has constructed microarchitectural executable formal models of some of its custom microprocessors in ACL2 [20]. The models have been integrated into a standard execution environment, replacing preexisting simulators written in more common programming languages such as C. The ACL2 models run at roughly the same speed as the original models. (How this is possible will become clear below). Reasoning about state machines requires symbolic simplification of terms representing states. Straightforward simplification algorithms can cause unnecessary exponential blowups in the size of the expression. This paper presents an algorithm for avoiding many of those explosions.

---

* This work was supported in part by Advanced Technology Center, Rockwell Collins, Inc., Cedar Rapids, Iowa.

2 The Problem

We present an algorithm for simplifying expressions that arise from the symbolic manipulation of formally described state machines. We use ACL2 term notation (i.e., Lisp notation). But the algorithm is of general interest in any formal setting where (a) terms are used to represent states, (b) "access" and "change" functions are provided, and (c) variable binding is present (e.g., Lisp `let` expressions, `lambda` applications, or, more generally, the application of defined functions). Our algorithm also deals with if-then-else constructs.

For example, a state, `s`, might have three components, named `a`, `b`, and `c`. We write `(a s)` to access the `a` component of `s` and `(update-a x s)` to create a new state like `s` but with `x` as its `a` component.

Of special interest are nests of updates. A simple example is shown below.

```
(let ((s (update-a (new-a x s) s)))                ; [*1]
  (let ((s (update-b (new-b x s) s)))
    (let ((s (update-c (new-c x s) s)))
      s)))
```

Each successive `let` changes the assignment of the variable `s`. So the `s` in the `new-b` expression refers to the state obtained by updating the `a` slot of the "original" (free) `s`.

Logically speaking, `(let ((`v_1 a_1`) ... (`v_n a_n`))` b`)` is equal to the instance of b obtained by simultaneously replacing all free occurrences of each v_i by the corresponding a_i. It is often read "let v_1 be a_1, …, and v_n be a_n in b," or perhaps more suggestively as "b, where v_1 is a_1, …, and v_n is a_n."

In ACL2, `let` expressions are syntactic sugar for certain `lambda` applications. Roughly speaking, `(let ((`v_1 a_1`) ... (`v_n a_n`))` b`)` is just `((lambda (`v_1 `...` v_n`)` b`)` a_1 `...` a_n`)`. We say "roughly speaking" because in ACL2 when we translate `let`s into `lambda` applications we make sure that every free variable of b is captured by the formal variables of the `lambda` (by adding extra formals and the corresponding actuals, as needed).

Replacing the `let`s in an expression by the corresponding `lambda` applications and performing beta reduction (i.e., expanding the `lambda`s away) may yield an exponentially larger term, because of variable duplication. This happens in `[*1]`.

We use `let` nests to describe state transformations as sequences of assignments to the components of the state. Formal models so expressed can be executed efficiently. The variable symbol `s` in `[*1]` is used in a "single-threaded" [5] way so that during execution on concrete data the original state may be destructively modified to create the new one. This efficiency is crucial to the use of the model as a simulator.

Now imagine defining a series of functions, e.g., `phase1`, `phase2`, …, in terms of expressions like `[*1]` and using them as the "updaters" in some `let` expression that produces a state. Realistic models involve many layers of definitions, culminating in some top-level state transition expression, e.g., `(machine x s)`.

We will present an algorithm for simplifying such expressions as `(b (machine x s))` with less computation than may at first appear necessary. One could do this by expanding away the `let`s, beta reducing all the `lambda`s and expanding all the (non-recursively) defined function applications, and then applying the obvious accessor/update rewrite rules, possibly in a "lazy" or outside-in way. However, the reader is urged to dismiss the thought that complete beta reduction (or the equivalent expansion of all non-recursively defined function definitions) is practical. Consider a C simulator for a system of interest and count the number of assignment statements: that is about the number of `let` bindings in the executable formal version of that model. Researchers at Rockwell Collins report [private communication]

> The typical complexity of high-level language models of these machine architectures has a depth around 300 assignment statements. That is, the execution of the simulator for one microcycle can involve the execution of about 300 state updates, which means that the translated-into-ACL2 model is a nest of state updates about 300 levels deep. Each "level" of the update nest typically contains at least two instances of state: the state being updated and a value being inserted typically expressed as a function of the state being updated.

If state is used twice at every level, the full beta reduction of such a term would contain on the order of 2^{300} occurrences of the updaters. From such considerations we conclude that it is impractical to contemplate full beta reduction of such models. We thus focus on simplification in the presence of such bindings.

3 Some Tests

Before presenting our algorithm we will present a simple test suite for it and show some performance data to motivate the rest of the paper. The simple test here is available at `http://www.cs.utexas.edu/users/moore/publications/nu-rewriter`.

In our simple test suite, we first declare a state object `s`, with two fields, `a` and `b`, accessed by functions of those names and updated by `update-a` and `update-b`. We next declare three uninterpreted function symbols, `v0`, `v1` and `v2`. Then we define `phase1` to do six successive updates on `s`, changing the `a` field to contain a new value computed conditionally as a function of the current `a` field using the three uninterpreted functions.

```
(defun phase1 (s)
 (let ((s (update-a
           (if (v0 1 (a s)) (v1 1 (a s)) (v2 1 (a s)))
           s)))
  (let ((s (update-a
            (if (v0 2 (a s)) (v1 2 (a s)) (v2 2 (a s)))
            s)))
    ...
     s...)))
```

Our first example, named `b-phase1`, is the theorem that `phase1` does not change the contents of the `b` field: `(equal (b (phase1 s)) (b s))`.

The second theorem, `b-phase1-phase1`, just composes `phase1` with itself, `(equal (b (phase1 (phase1 s))) (b s))`, and could be proved trivially from `b-phase1` except that we prevent such a proof by disabling `b-phase1`.

The third theorem, `a-phase1`, describes the value of the `a` field after `phase1`.

We then complicate the test by defining two more phases. `Phase0` copies the `a` field into the `b` field. `Phase2` copies the `b` field into the `a` field. We define `machine` to do `phase0`, then two `phase1` steps, and then `phase2`.

The fourth theorem, `a-machine`, shows that `machine` does not change the `a` field, `(equal (a (machine s)) (a s))`. The fifth, `b-machine`, shows that the final `b` field is the initial `a` field, `(equal (b (machine s)) (a s))`.

Each theorem can be proved by rewriting alone. We prove each with ACL2 Version 2.6 (the first to include our algorithm) in each of two configurations. In "standard ACL2," the algorithm is disabled; in "ν-ACL2," the algorithm is enabled. All of the tests were conducted running under Allegro Common Lisp on a 731 MHz dual-processor Pentium III. Time is measured in seconds. The results are shown in Figure 1.

Theorem	standard ACL2	ν-ACL2
`b-phase1`	0.48	0.01
`b-phase1-phase1`	128.76	0.01
`a-phase1`	0.41	0.04
`a-machine`	139.39	0.02
`b-machine`	143.91	0.02

Fig. 1. Seconds to Prove Theorems on 731 MHz Pentium III

Note the growth in standard ACL2's times from `b-phase1` to `b-phase1-phase1`. Comparing the old rewriter's performance with that of the new one on industrial data is essentially impossible because the old rewriter exhausts resources before completing interesting problems of the kind handled routinely by the improved system. (Adding one more `phase1` step to `b-phase1-phase1` causes standard ACL2 to exhaust memory after six hours of computation; ν-ACL2 does "`b-phase1`^{3}" in 0.11, `b-phase1`^{4} in 6.46, and `b-phase1`^{5} in 412 seconds.)

The terms arising in typical machine models are not as regular as those in this test suite. Our algorithm does not distinguish "control" from "data," require the identification of "phases," or limit itself to single-threaded states. In addition, typical industrial machine states have hundreds of components. Some of those components are atomic (e.g., contain booleans, integers, etc.) others may themselves be structured as records or arrays. ACL2 supports states containing arrays and the simplification algorithm we have implemented does also. But in this paper we confine our attention to "flat" states.

4 Terminology

We now prepare to describe our algorithm precisely, starting with the terminology and conventions we use. In ACL2, `let` expressions are just syntactic sugar for `lambda` applications. `Lambda` expressions are handled just like other function applications. Each `lambda` expression has a list of *formal variables* and a term for a *body*. All free variables in the body are among the formals. Functions may only be applied to the correct number of actuals. The function application (f a_1 ... a_n) is equal to its beta reduction, the result of instantiating the body of f with the substitution replacing v_i by a_i. We use the verbs "to open" or "to expand" to describe the replacement of a function application by its beta reduction. If f is a `lambda` expression or f is a function symbol and that symbol is not used as a function symbol in the body of f, we say f is *non-recursive*. Henceforth, we do not talk formally about `let`s but about non-recursive function applications.

In ACL2 the state accessor and updater functions are logically defined in terms of a "universal" accessor `nth` and a "universal" updater, `update-nth`, where `(nth i s)` extracts the `i`th element of the list `s` and `(update-nth i v s)` constructs a list like `s` but whose `i`th element is `v`. Thus, a term like `(b (update-c x s))` expands to `(nth 1 (update-nth 2 x s))`. Our algorithm is fundamentally concerned with applying the theorem

Theorem. `nth-update-nth`:

```
(equal (nth i (update-nth j v s))
       (if (equal (nfix i) (nfix j)) v (nth i s)))
```

as a rewrite rule (left-to-right). The function `nfix` is the identity on natural numbers and otherwise is 0. Its use in the theorem above is a reflection of the absence of syntactic typing in the language. The theorem says that the `i`th component of the state produced by updating the `j`th component of `s` with `v` is either `v` or the `i`th component of `s`, depending on whether `i` and `j` are equal. The definitions of user-level state access/update functions (e.g., `b` and `update-c`) are treated as ordinary function definitions like `phase1` above.

We call expressions like `[*1]` "`nth/update` expressions" or *ν-expressions* (for "nu" or "`nth/update`"). This loosely defined class of expressions includes state accessor/updater functions defined in terms of `nth` and `update-nth`, their array counterparts, if-then-else expressions, and variable binding constructs such as `let` or function or `lambda` application.

5 Binding Stacks, Facets, and Reconciliation

ACL2's standard rewriter is inside-out. To rewrite (f a_1 ... a_n) it first rewrites the a_i to standardize them. Thus, the opportunity to apply `nth-update-nth` to `(b (phase1 x s))` occurs only after `(phase1 x s)` is expanded to an `update--nth` expression. This may exponentially increase the size of the term.

Instead of rewriting a_2 in (`nth` a_1 a_2) we wish to "look ahead" to see whether we can "see" a_2 as an `update-nth` expression by expanding non-recursive functions. For example `[*1]` can be seen as an `update-c` expression, which can, in turn, be seen as an `update-nth` expression. These expressions must be understood in an appropriate variable binding environment. Note that the `update-c` expression in `[*1]` buried in the expression and would be the late in the process of ordinary rewriting. By `nth-update-nth`, if the indices in the `nth` and `update-nth` expressions are the same, the answer is `(new-c x s)`, under appropriate bindings for `x` and `s`; if the indices are unequal, the answer is (nth a_1 s), under appropriate bindings. Clearly, if we can decide the equality of the indices then work can be saved. (Often, in this setting, the indices are constants.) The challenge is to keep the bindings straight.

Many applications require descending through hundreds of `lambda` expressions. We want to "be" inside the deepest `lambda` without creating the instance. We therefore introduce the idea of seeing a term in the context of a substitution and we represent the substitution as a stack of function call frames. This is just a generalized version of a nest of `lambda` applications. We call this object a "facet" and define it below.

A *binding stack* is a stack of frames. Each frame contains a list of n variables and a list of n terms. The free variables occurring in the terms of a frame (other than the deepest frame) are among the variables of the frame immediately below.

We represent stacks as lists, where the first element of the list is the top frame. Here is a stack containing two frames,

```
(((a b)   . ((afn u w) (bfn u v)))             ; frame 1
 ((u w v) . ((ufn s) (wfn s) (vfn s)))).       ; frame 2
```

Call this stack σ. In the top frame of σ, frame 1, `a` is associated with `(afn u w)` and `b` with `(bfn u v)`. We say `(afn u w)` is the term *corresponding* to `a` in that frame. The representation of frames this way, rather than as association lists, makes them faster and cheaper to create.

A stack *represents* the substitution created by pairing each variable in the top frame with the result of instantiating its corresponding term with the substitution represented by the rest of the stack. Thus, the stack σ represents the substitution that replaces `a` by `(afn (ufn s) (wfn s))` and `b` by `(bfn (ufn s) (vfn s))`.

A *facet* is a pair consisting of a term t and stack σ, written $< t, \sigma >$, and represents the instance of t under the substitution represented by σ. Hence, if σ is the example stack above, the facet $<$`(h a b)`,$\sigma >$ represents `(h (afn (ufn s) (wfn s)) (bfn (ufn s) (vfn s)))`.

When we refer to a facet as though it were a term, we mean to refer to its term component. An *empty facet* is one whose stack is the empty list, `()`.

The function symbol of a non-variable, non-constant facet is the same as the function symbol of the term it represents. This allows us seldom to create the substitutions represented by stacks or the terms represented by facets. Instead, we "chase" the variable bindings when we need them. Facets are similar to the records and binding environments of the structure sharing representation of clauses [4]. Another way to think of a facet is that it is a nest of `lambda` applications turned inside out and flattened. Given a nest of `lambda` applications, the term of the corresponding facet is the body of the innermost `lambda` expression and the stack of the facet is the list of paired formals and actuals, starting with that for the innermost `lambda` application and proceeding outwards. Facets have two computationally convenient properties. First, if the term of a facet is an application of a defined non-recursive function, then we can represent the expansion of that function by a facet easily derived from the first. Second, if the term of a facet mentions a variable symbol then we can easily find out how that variable symbol is replaced by the substitution and we can represent the actual expression by another facet easily derived from the first. `Lambda` expressions are nested the "wrong way" to make these operations efficient.

We define finite chains of facets related by a generalized notion of expansion. Let ϕ be the non-empty facet $< t, \sigma >$. Then its *expansion*, ϕ', is defined as follows. If t is a variable symbol that is not a member of the variables in the top frame of σ or t is a constant, ϕ' is $< t, \texttt{()} >$. If t is a variable symbol that is a member of the variables in the top frame of σ, ϕ' is $< t', \sigma' >$, where t' is the term corresponding to t in the top frame and σ' is the result of removing the top frame from σ. If t is the application of a defined non-recursive function, f, with formals $\boldsymbol{v}$ and body b, to actual expressions $\boldsymbol{a}$, ϕ' is $< b,\ \texttt{((}\boldsymbol{v}\ \texttt{.}\ \boldsymbol{a}\texttt{)}\ \texttt{.}\ \sigma\texttt{)}>$, i.e., the facet whose term is the body of f and whose stack is obtained from σ by pushing a new frame containing the formals and actuals. In all other cases no expansion is possible.

All the facets in an expansion chain represent equal terms. We call them "facets" because they are different ways of looking at a term.

The *preferred* facet of a facet ϕ is the last facet in its expansion chain. Note that since `update-nth` is a recursive function, if ϕ can be seen as an instance of an `update-nth` term by sufficient expansions of non-recursive functions, then the preferred facet of ϕ will have an `update-nth` term as its term component.

Given a facet we can economically create a term equal to the one it represents, using `lambda` abstraction. The `lambda` *abstraction* of the facet $< b, \texttt{()} >$ is the term b. The `lambda` *abstraction* of $< b, \texttt{((}\boldsymbol{v}\texttt{.}\boldsymbol{a}\texttt{)}\ \texttt{.}\ \sigma >$ is the `lambda` abstraction of $<\texttt{((lambda}\ \boldsymbol{v}\ b\texttt{)}\ \boldsymbol{a}\texttt{)}, \sigma >$. Note the bindings of the abstraction occur in the opposite order. The size of the `lambda` abstraction of a facet is linear in the size of the facet.

An important optimization of `lambda` abstraction is to eliminate unnecessary bound variables. If the body of a `lambda` does not use a variable symbol that is listed in the formals, it and the corresponding actual can be eliminated. Another optimization is that variables bound to constants can be eliminated.

Because we will manipulate facets in lieu of the terms they represent, we will also have occasion to form new facets by putting together several others.

For example, let ϕ_i, $1 \le i \le n$, be n facets, each of the form $< t_i, \sigma_i >$. Each ϕ_i represents a term r_i. Think of the ϕ_i as having been generated by applying our algorithm to the arguments of a call of some function f. We wish to represent the term $\texttt{(}f\ r_1\ \ldots\ r_n\texttt{)}$ as a facet. We call this the *reconciliation* of $\texttt{(}f\ \phi_1\ \ldots\ \phi_n\texttt{)}$. Note that $\texttt{(}f\ \phi_1\ \ldots\ \phi_n\texttt{)}$ is neither a term nor a facet. It fails to be a term because it contains facets. It fails to be a facet because there is no single, outermost stack.

The reconciliation of $(f\ \phi_1\ \ldots\ \phi_n)$ is computed as follows. We first find the greatest common ancestor stack, σ, of the σ_i. Let ρ_i be the top part of σ_i, down to the common ancestor σ. Thus, σ_i is the concatenation of ρ_i and σ. Let t'_i be the `lambda` abstraction of the facet $< t_i, \rho_i >$. Then $<(f\ t'_1\ \ldots\ t'_n), \sigma >$ is the reconciliation of $(f\ \phi_1\ \ldots\ \phi_n)$ and is a facet that represents a term equal to $(f\ r_1\ \ldots\ r_n)$.

Reconciliation has two important optimizations. The first is that preferred constant facets, i.e., facets whose terms are constant expressions, have empty stacks. If these empty stacks participate in the greatest common ancestor computation, the ancestor stack is always `()`, meaning the reconciled subexpressions share no subterms. But constants denote themselves in any stack. So we ignore constant facets when determining the ancestor. The second optimization of reconciliation exploits an empirical observation. Frequently all the non-constant facets in a reconciliation have the same stack. In that case, that stack is the ancestor. This case arises so frequently (in 98% of the cases over a test involving roughly 100,000 reconciliations) that it is worthwhile to code for it.

6 Our Algorithm

We now describe an algorithm for simplifying a term by applying `nth-update--nth` and expanding functions. We call the rewriter the "ν-rewriter." The algorithm operates on facets. To use it on terms we apply it to the empty facet containing the term and then we `lambda` abstract the resulting facet.

The ν-rewrite Algorithm

1. We wish to ν-rewrite the facet ϕ. Let ϕ' be the preferred facet of ϕ. If ϕ' is a variable or constant facet or the term of ϕ' does not begin with `nth`, we return ϕ'.
2. Otherwise, ϕ' is $<$`(nth` $i\ t$`)`$, \sigma >$. Let $\hat{i}$ be the facet obtained by ν-rewriting $< i, \sigma >$. Let $\hat{t}$ be the preferred facet of $< t, \sigma >$. If $\hat{t}$ is a variable or constant, we reconcile and return `(nth` $\hat{i}\ \hat{t}$`)`.
3. At this point, we know $\hat{t}$ is a function application. Since $\hat{t}$ is a preferred facet, its term is not a `lambda` application. Let f be the function symbol of $\hat{t}$. Our code considers five cases on f: it is `if`, `update-nth`, `update-nth-array`, `nth`, or some other symbol.
   - 3.1 If f is `if`, then $\hat{t}$ is of the form $<$`(if` $a\ b\ c$`)`$, \rho >$. Let ϕ_1 be the result of reconciling and ν-rewriting `(nth` $\hat{i}\ < b, \rho >$`)` and let ϕ_2 be the result of reconciling and ν-rewriting `(nth` $\hat{i}$ $< c, \rho >$`)`.
     - 3.1.1. If ϕ_1 and ϕ_2 are the same facet, return ϕ_1.
     - 3.1.2. If no applications of `nth-update-nth` were made in producing ϕ_1 or ϕ_2, then return the reconciliation of `(nth` $\hat{i}\ \hat{t}$`)`.

3.1.3. Otherwise, let ϕ_0 be the result of ν-rewriting $< a, \rho >$.

3.1.3.1. If ϕ_0 is a constant facet, return ϕ_2 or ϕ_1 according to whether the constant is `nil` (i.e., the test of the `if` can be decided).

3.1.3.2. Otherwise, return the reconciliation of (`if` ϕ_0 ϕ_1 ϕ_2).

3.2. If f is `update-nth`, $\hat{t}$ is of the form $<$(`update-nth` j v s)$, \rho >$. Let $\hat{j}$ be the result of ν-rewriting the facet $< j, \rho >$.

3.2.1. If $\hat{i}$ and $\hat{j}$ represent equal naturals, we return the result of ν-rewriting the facet $< v, \rho >$.

3.2.2. If $\hat{i}$ and $\hat{j}$ represent unequal naturals, we return the result of ν-rewriting the reconciliation of (`nth` $\hat{i}$ $< s, \rho >$).

3.2.3. Otherwise, we return the reconciliation of (`if (equal (nfix` $\hat{i}$`) (nfix` $\hat{j}$`))` $< v, \rho >$ (`nth` $\hat{i}$ $< s, \rho >$)).

3.3 and 3.4. If f is either `update-nth-array` or another `nth`, then (assuming the original term was derived from a state access/update nest) we are dealing with an array or some other structured component. To keep this paper brief, we do not discuss that case here, but it is analogous to what we have described.

3.5. If f is some other symbol, then we return the reconciliation of (`nth` $\hat{i}$ $\hat{t}$).

7 Discussion

The algorithm focuses entirely on terms of the form (`nth` i t). The main case split is on the form of t.

In paragraph 3.1 we consider the case that t can be seen as an if-then-else expression. We might be ν-rewriting a term like (`nth` i (`if` a b c)), but more often we are ν-rewriting a term like (`nth` i (`phase` a s)), where `phase` is defined to be a nest of `let`s with an `if` expression as the body.

Observe that in attacking (`nth` i (`if` a b c)) we first "distribute" the `if`, moving the `nth` onto b and c. After rewriting these two subgoals we ask whether the resulting facets are equal. If so, we can avoid rewriting a by virtue of `(if x y y)` = `y`. Of course, we might have chosen to rewrite a first and determined that it is equal to `nil`, say, thereby avoiding the need to rewrite b. But the ν-rewriter has relatively little support for deciding propositions (since it is context free and does not use the ACL2 type system or other decision procedures).

To see why the "`(if x y y)`" heuristic so often wins, consider the origins of the problem. Here b and c are state transformations, the modeled machine is branching on a, and we are interested in determining the i^{th} component of the new state. But most state transformations on the machines we have seen leave most state components unchanged. Thus, in many cases neither b nor c change the value of the i^{th} component and our heuristic makes the superior choice.

In paragraph 3.1.2 we basically abandon the rewriting of `(nth` i `(if` a b c`))` if no `nth-update-nth` rule was applied while rewriting `(nth` i b`)` or `(nth` i c`)`. We prefer to keep the `if` inside the `nth` to avoid case splitting. To implement the test, the ν-rewriter returns a flag that indicates whether it used any rules. It is insufficient to test whether the rewritten facets are equal to their unrewritten versions since quite often b and c will have been replaced by their preferred facets (i.e., we may have opened function applications).

Paragraph 3.2 is the case for which the algorithm was invented. It applies the `nth-update-nth` theorem.

Paragraphs 3.3 and 3.4 deal with arrays in our setting and are not discussed here.

We have optimized the algorithm in several ways. The most important is to use caching or memoization to avoid recomputing the ν-rewrite of a previously seen facet. In our implementation, we use a hash table with 64K entries, each of which is a ring containing (at most) the five most recently seen facets that hashed to that location and the results of the corresponding ν-rewrites. Even though we hit on a hash entry only approximately 6% of the time, we find that the savings is significant and, indeed, makes the difference between being practical or impractical on industrial-scale problems.

Recall the tests in Section 3. Consider the theorem there called `b-phase1--phase1`. Implementing the algorithm without caching gives rise to 10,236 calls of the ν-rewriter. With caching, that theorem generates 124 calls. Of those 124 calls, 18 hit in the cache, giving a cache hit rate of 14%. Each hit, however, saves the algorithm from re-exploring a potentially large subtree.

In practical applications, the cache is of supreme importance. For example, in a theorem taken from the proprietary Rockwell test suite, the cached version of the ν-rewriter was called 216,524 times. The cache hit rate was 6.2%. But without the cache the algorithm would require about 3×10^{26} calls.[1]

Because of our desire to cache the results, we have made the ν-rewriter completely "context-free." That is, it does not take any arguments that encode the hypotheses governing the current term, since to do so would mean that we would have to cache that contextual information and probably have to probe the cache to look for prior calls in weaker contexts rather than identical contexts.

For a discussion of several elaborations of the algorithm, how it is used in ACL2's rewriter, and some proposed improvements, see `http://www.cs.-utexas.edu/users/moore/publications/nu-rewriter`.

8 Related Work

A term representation similar to our facets is provided by the "term module" of Hickey and Nogin's modular theorem proving architecture [10]. Their notion of "delayed substitution" is motivated by the same considerations that led us to

[1] The number is 338,664,298,746,582,325,860,641,409. This is too large compute by the brute force method of eliminating the cache and counting calls. It was computed by using the cache to remember how much work was done for each entry.

introduce facets. Their framework is more general than ours; in particular, they provide utilities for fast tactic-based theorem proving. However, their approach to delayed substitution is, essentially, to use `lambda` applications to represent terms and to implement the operations of destructuring such terms without doing the substitution implied by beta reduction. Our facet data structure is more efficient for the operations we support. This is important when dealing with very deep `lambda` nests.

Our notion of reconciliation, which is designed to generate a facet from a term-like structure containing facets, has no counterpart in their system because their "facets" are already terms. We can afford reconciliation because, as noted, about 98% of the time the facets to be reconciled all have the same stack.

The architecture of [10] does not provide caching, which we have found is crucial to good performance on large problems.

Facets are suggestive but independent of "explicit substitution" logics [7, 1, 2]. Our view of facets is that they merely provide an efficient data structure for implementing certain simplification strategies in conventional logics. The idea of "nameless" substitutions might be usefully incorporated in future work.

9 Conclusion

Our algorithm is being tested under fire in industrial applications. We are still "tuning" our integration of the algorithm, focusing on tactics for using it and certain low-level implementation details. Of particular interest are the management of the cache and the associated hashing function used to cache Lisp s-expressions. The algorithm sometimes generates unnecessarily large intermediate expressions as suggested by the `b-phase1`^{i} series mentioned in Section 3. We are working on preventing these explosions

Nonetheless, the ν-rewriter has been extremely effective in the full-scale industrial application for which it was developed for Rockwell Collins. It has been used in the proofs of hundreds of theorems that were previously well beyond the capability of ACL2 to simplify. We take this as a good sign but still regard this as a work in progress.

10 Acknowledgments

I thank Dave Greve and Matt Wilding of the Advanced Technology Center of Rockwell Collins for their inspiration and support of this idea. I also thank Mark Bickford, Matt Kaufmann, Pete Manolios, and Matt Wilding for their contributions to this paper.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lvy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

2. M. Ayala-Ricon and Cesar Munoz. Explicit subsitutions and all that. Technical Report TR-2000-45, ICASE, NASA Langley Research Center, Hampton, Virginia, November 2000. `http://www.icase.edu/Dienst/UI/2.0/Describe/ncstrl.icase/TR-2000-45`.
3. W.R. Bevier, W.A. Hunt, J S. Moore, and W.D. Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.
4. R. S. Boyer and J S. Moore. The sharing of structure in theorem-proving programs. In *Machine Intelligence 7*, pages 101–116. Edinburgh University Press, 1972.
5. R. S. Boyer and J S. Moore. Single-threaded objects in ACL2. *(submitted for publication)*, 1999.
6. Bishop Brock and Warren A. Hunt, Jr. Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP. In *1997 IEEE International Conference on Computer Design*, pages 31–36. IEEE Computer Society, October 1997.
7. N. G. de Bruijn. A namefree lambda calculus with facilities for internal definition of expressions and segments. Technical Report TH-Report 78-WSK-03, Department of Mathematics, Technological University Eindhoven, Netherlands, 1978.
8. M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
9. J. Grundy. Verified optimizations for the intel ia-64 architecture. In *TPHOLs 2000, LNCS 1869*, pages 215–232. Springer-Verlag, 2000.
10. J. Hickey and A. Nogin. Fast tactic-based theorem proving. In *TPHOLs 2000, LNCS 1869*, pages 252–267. Springer-Verlag, 2000.
11. W.A. Hunt and B. Brock. A formal HDL and its use in the FM9001 verification. *Proceedings of the Royal Society*, April 1992.
12. M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Press, 2000.
13. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, 2000.
14. P. Manolios. Correctness of pipelined machines. In *Formal Methods in Computer-Aided Design, FMCAD 2000*, pages 161–178. Springer-Verlag LNCS 1954, 2000.
15. S. P. Miller and M. Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *Proceedings of WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16. IEEECS, April 1995.
16. J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language.* Automated Reasoning Series, Kluwer Academic Publishers, 1996.
17. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Lecture Notes in Artificial Intelligence, Vol 607, Springer-Verlag, June 1992.
18. J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In *Computed Aided Verification, CAV '98*, pages 135–146. Springer-Verlag LNCS 1427, 1998.
19. G. L. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.
20. Matthew Wilding, David Greve, and David Hardin. Efficient simulation of formal processor models. *Formal Methods in System Design*, to appear. Draft TR available as `http://pobox.com/users/hokie/docs/efm.ps`.