Factotum: Automatic and Systematic Sharing Support for Systems Analyzers

David James Sherman and Nicolas Magnier

Laboratoire Bordelais de Recherche en Informatique
CNRS/Université Bordeaux-1, Bordeaux, France
sherman|magnier@LaBRI.U-Bordeaux.FR

Abstract. Tools for systems analysis often combine different memory-intensive data structures, such as BDDs, tuple sets, and symbolic expressions. When separate packages are used to manipulate these structures, their conflicting resource needs can reduce overall performance despite the individual efficiency of each. *Factotum* is a software system for implementing symbolic computing systems on DAG-based structures that critically rely on sharing of equivalent subterms. It provides an subterm sharing facility that is *automatic* and *systematic*, analogously to the way that automatic memory management is provided by a garbage collector. It also provides a high-level programming interface suitable for use in multithreaded applications. We describe both the theoretical underpinnings and practical aspects of Factotum, show some examples, and report on some recent experiments.

1 Introduction

The size of in-core representations of complex systems is one of the important performance issues for model-checking and other systems analysis tools. *Sharing* of computationally equivalent substructures is a critical need for reasonable performance in these applications: not only does it provide *compact* representations, permitting the entire system being analyzed to be loaded into memory and processed, it is the key to avoiding unnecessary recomputation of equivalent properties.

While automatic garbage collection[McC60,Wil95] has become an accepted practice for managing memory-intensive applications, automatic sharing management is significantly less developed. Sharing of equivalent data structures in memory is well-developed from an algorithmic standpoint, just as explicit allocation and deallocation is an important subject of algorithmic considerations. But uniformly-applied *automatic* detection and maintenance of sharing, analogous to automatic garbage collection, is rarely studied as a general-purpose technique.

Two clear cases where automatic sharing is both critical and well-developed, especially in the context of systems analysis, are *binary decision diagrams*[Ake78] [Bry86,Rau96] and *sharing trees*[ZC95a]. Implementations of binary decision diagrams guarantee, either systematically (for ROBDDs) or at regular intervals

(for QROBDDs), that all equivalent subgraphs are fully shared in a DAG structure. The result is a compact and canonical representation of a truth table, that permits efficient calculations that would otherwise explode in time and space. Sharing trees are efficient DAG representations of tuple sets, where maximal prefix and suffix sharing gives a compact and canonical representation that often admits efficient operations.

A third domain in which sharing of equivalent structures has proved important is term rewriting, where sharing induced by the use of congruence closure and its variants can result in exponential speedups at modest (quadratic or linear) cost[Che80,RV90,She94,Mag94,Ver95]. These techniques have not, however, made their way into the mainstream of accepted practice. The same argument can be made for memo functions[Mit68] in functional programming languages[Hug85]; while memoization (also called *tabulation*) is well-known, it is not systematically applied to all functions.

Factotum is an attempt to provide the advantages of automatic uniform sharing to implementors of systems analyzers in general, where computation is performed on labeled tree-like structures with sharing.[1] It grew out of the needs of the Clovis project[2], where decision-diagrams, sharing trees, and memoized function evaluation are needed simultaneously. Factotum is directly inspired by automatic garbage collection, and aims to offer the same kind of freedom from implementation details and the same kinds of guarantees about correctness and global performance.

1.1 Key Observations

The key observation is that the notions of equivalence that permit sharing in the systems cited above are congruences. That is, the addresses of nodes in memory can be considered as names of equivalence classes, and any two nodes with the same label and pairwise equivalent children are also equivalent. Two children are known to be equivalent if they have the same equivalence class name, that is, if their addresses are equal. BDDs and sharing trees use hash tables to discover that an equivalent node already exists and can be shared. While these systems guarantee prefix sharing by construction, suffix sharing is introduced by a *reduce* operation that consults this hash table.

It should consequently be possible to generalize the congruence-based techniques from term rewriting to provide a system of more widespread utility.

Several other observations, based on experience with the systems described above and analysis of their implementation, motivated the design of Factotum.

1. Efficiency in these systems is due to sharing of equivalent nodes.

[1] We use the slightly odd term "tree-like" to describe structures that are trees from a naive algorithmic standpoint but are concretely implemented as directed acyclic graphs. Terms (with sharing) are one example.

[2] A collaboration between the LaBRI (Université Bordeaux-1) and the LIP-6 (Université Paris-6) laboratories, concerning modular model-checking of industrial systems.

2. Lack of sharing means a loss of efficiency,[3] but not of correction.
3. Sharing is (re-)established by calls to a *reduce* operation throughout the code.
4. The algorithms for the basic operations are defined at a low level of abstraction.
5. The necessary top-down and bottom-up traversals of the data structures, used to implement basic operations , are defined ad hoc.

1.2 Key claims

The key claims that we make based on these observations are the following.

First, every one of the systems we have studied implements sharing discovery and maintenance in its own way. There is therefore a lot of reimplemention and revalidation, of what is perhaps the least interesting and most fastidious part of the system.

Second, *optimal* algorithms do not necessarily mean *fast* implementations, unless the measure of optimality considers the sequence of operations that will be performed. While one can show that the set operations in [ZC95a] are optimal in the sense that each restores maximal sharing when it is done, this requirement imposes a systematic overhead, and it is not clear whether overall performance might not be improved in some cases by checking for sharing less often. A copying garbage collector can improve overall performance when most objects have short lifetimes[App87,Wil95], simply because it is less eager reclaiming memory; in the same way, a basic sharing service may improve overall performance by checking for sharing less frequently.

Third, and finally, sharing ought to be *automatic* and *systematic*. It should be available without the need for algorithmic support in the application code, and it should be available for all objects manipulated by the system.

The Factotum system aims to provide just such an automatic and systematic sharing service, validated once and for all, where the sharing policy can be fine-tuned for overall performance, and where the structures in memory are manipulated using high-level tools. Factotum makes it possible to integrate, in the same application, different system representations and analysis techniques while guaranteeing overall correctness and performance.

1.3 Outline of the paper

In the first part of what follows, section 2, we give a brief overview of the Factotum system and its basic concepts. Section 3 gives the theoretical foundations and underlying semantics of the system, using the *RWS calculus*. Some concrete and intuitive examples of Factotum use are shown in section 4, while section 5 reports on our initial experiments. We conclude in section 6.

[3] Perhaps catastrophically.

2 Factotum Concepts

Factotum is conceived as a programmer's toolbox, providing an integrated set of operations on tree-like structures with sharing. The tools provided by Factotum are normally used by higher-level application code to perform some kind of symbolic computation, such as set-based or term-rewriting calculations.

The applications programmer attaches labels to the tree-like structures provided by Factotum, and assigns an *application-specific* semantics to these labeled structures. Factotum does its best to provide an efficient in-core representation, including maintaining and introducing sharing of equivalent structures. This approach maintains a clear separation between *mechanism* and *policy*: Factotum provides the means to perform operations and some basic guarantees, while the client decides what operations to perform and what the result means.[4] Structures with different semantics can be freely mixed in memory.

The runtime system is automatically initialized at the first use of a Factotum object. Memory and sharing management take place automatically during the course of the computation.

The sharing subsystem in Factotum works independently from the garbage collector to share nodes in memory when possible and convenient. It can, like the garbage collector, intervene at any moment during the computation. Sharing is transparent: the application cannot detect the presence (or absence) of sharing for a given node. Whether it is possible to share two given nodes is determined by node equivalencies indicated by the application code, and observed congruence between existing nodes. The fact that two nodes can be shared does not mean that they are, or will be. The sharing subsystem determines when it is advisable to share nodes based on its own internal policies and its own history of sharing.

2.1 Names and Cursors

The basic Factotum objects that the applications programmer manipulates are `Names`, that contain fixed references to nodes in memory, and `Cursors`, that encapsulate an arbitrary exploration below a fixed reference. Figure 1 summarizes the relations between nodes in memory and the objects that refer to them.

A cursor encapsulates an arbitrary tree-like inspection of a structure in memory. A `Cursor` object always refers to some node in memory, and, like `Name` objects, can inspect or modify the label, arity, and children of the node. A cursor can also be moved up, down, right, and left in the tree, relative to its current position. A cursor is restricted to the subtree rooted at the node at which the cursor was created.

However, the meaning of cursor movement operations is not clear when nodes may be arbitrarily shared. Figure 2 shows the confusion in the definition of the parent and sibling relations between nodes that arises in the presence of uniform sharing.

[4] This is a longstanding distinction, of which the X Window System remains one of the most popular proponents. An interesting recent discussion of the difference between *strategy*, *policy*, and *mechanism* can be found in [WJNB95].

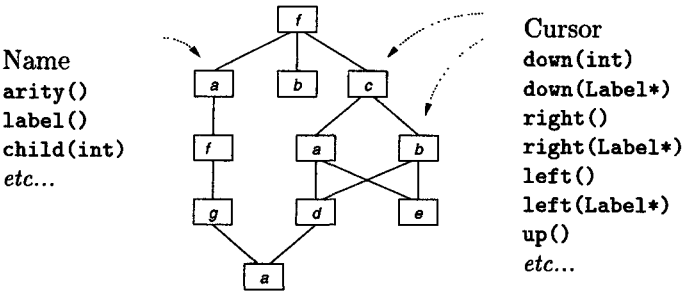


Fig. 1. `Name` and `Cursor` objects are the two ways to refer to nodes in memory

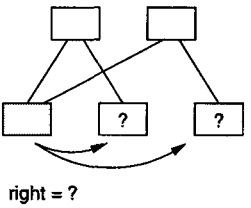


Fig. 2. Ambiguity in the right-sibling relation due to sharing

The solution is that *every cursor remembers the history of its descent* in the data structure. It is in this way that a cursor is said to encapsulate its exploration. Since the parent relations between nodes are established relative to the unique history of a cursor, there is no ambiguity in the choice of the parent and siblings of a node in a given context.

This solution also has two further practical consequences. First, since these relations are defined relative to the stored histories of cursors, the data structure in memory has *no provision for storing parent and sibling pointers.* This greatly reduces the physical size of the objects in memory, allowing us to store larger structures, and greatly reduces the need for costly pointer update when nodes are added and deleted. Second, the encapsulation of the state of a cursor lends itself to mutithreaded applications, where only the global data structure in memory is shared between processes.

2.2 Mementos

The state of a cursor can be stored and restored at a later time by a memento object. The state encapsulated in such a memento is defined by the cursor history and the node the cursor presently refers to. A memento is stored and discarded by the application program, and can be applied an arbitrary number of times.

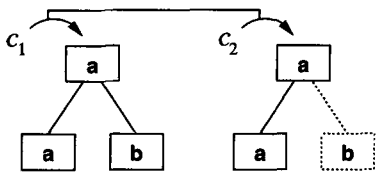
Among other things, mementos provide an *undo* facility. When a memento is applied, any modifications to the cursor since the creation of the memento are forgotten, and any nonpersistent modifications to the subject data structure are abandoned.

2.3 Coupled Traversals

An important observation is that most interesting analyzer operations are *simultaneous*, or parallel, traversals of different structures. Consider the case of set difference in sharing trees[ZC95a]. The basic algorithm is to trace the paths in the minuend and the subtrahend, and only include in the result those paths that are in the one but not the other. Another example is tree pattern-matching[HO82], which compares a pattern term to a subject term, and responds with a match when the two agree. In both cases the intuitively satisfying solution is to define traversals with several cursors moving together. Factotum provides specialized support for these kinds of applications: *coupled traversals.*

In a coupled traversal, a vector of cursors is moved in parallel. The first component, the *independent cursor*, is moved by the traversal as it would be normally. The other components, the *dependent cursors*, try to follow the independent cursor to nodes in their own data structure with the same labels. The traversal also maintains a vector of status registers; dependent cursors that were not able to follow the independent cursor are marked *invalid*, and remain so until the traversal brings them back up to a point where they agree with the independent cursor again.

Oftentimes we want to detect the disagreement of a dependent cursor, make some change to the data structure, and retry the move to bring the cursor back to a valid state. The `retry` operation of the traversal does exactly that.



Cursor c_1	status	Cursor c_2	status
`down 1`	valid	`down(a)`	valid
up	valid	up	valid
`down 2`	valid	`down(b)`	invalid
		`add_child`	invalid
		`set_label(b)`	invalid
		`retry`	valid

Fig. 3. Coupled traversal of two structures

Figure 3 shows an example of two coupled cursors c_1 and c_2, where the second lacks a child labeled **b**. The figure shows the sequence of operations emitted by the traversal, intermixed with the client operations that add the necessary child when the dependent cursor becomes invalid.

3 Underlying Semantics

Factotum is more than a simple memory manager, since it constantly *improves* the representation in memory of the data it manages. Three natural questions immediately impose themselves: What improvements to the data structure in memory are permitted? Who can perform them? What justifies that they are correct?

In short, we need to characterize the *permissible transformations* of the data structures in memory, and define an *underlying semantics* that justifies the improvements. This underlying semantics must admit sharing from congruence, so that we can respond to the requirements evoked in sections 1.1 and 1.2.

3.1 The RWS Calculus

Let Name be a primitive domain of variables, and Sig be a domain of shallow terms of depth at most two over some alphabet Σ with rank function ρ and the variables in Name. We write, for example, $X, Y, Z \in$ Name and $a, g(X), f(X_1, X_2) \in$ Sig. The RWS calculus[She90,She94] uses three tables to define the state of a rewriting system:

bind : Name $\rightarrow$ Sig $\cup$ {Nil}
reduce : Name $\rightarrow$ Name $\cup$ {Nil}
index : Sig $\rightarrow$ Name $\cup$ {Nil}

Formally, congruence is defined by the following.

Definition 1. *Let E be a binary relation on elements of* Name. *Two variables $X, Y \in$ Name are* congruent *under E if bind $X = f(X_1, ..., X_{\rho f})$; bind $Y = f(Y_1, ..., Y_{\rho f})$; and X_i E Y_i for all i from 1 to ρf.*

For any binary relation E on elements of Name, we can define *congruence under* E.

Let *reduce** be the closure of *reduce*. Of the different interpretations of these tables, the useful one in this context is given by the function *rterm*:

Definition 2. *Function rterm* : Name $\rightarrow \Sigma^{terms}$ *produces the term given by following reduce entries as far as possible for each subterm:*

$$rterm\ X = \begin{array}{l} f(rterm\ X_1, \ldots, rterm\ X_{\rho f}) \\ \text{where } bind\ reduce^*\ X = f(X_1, \ldots, X_{\rho f}) \end{array}.$$

It is useful to consider an example of how we can use these tables. Suppose we want to represent all of the pertinent information about the state of an ordinary term-rewriting system, with labeled terms and the usual sort of rewriting rules. Consider an initial term $f(d, h(b))$ subject to the rules $\{b \rightarrow a, h(a) \rightarrow c, d \rightarrow g(h(a))\}$. The left half of figure 4 on page 9 shows the contents of the three tables after rewriting this term, and the right half gives a schematic interpretation of the table contents. Note that $X_2 = c$, $X_2 = h(a)$, and $X_2 = h(b)$ are logical consequences of the table contents. The three tables represent a state of

knowledge about a rewriting system. The RWS calculus allows transformations between such states by means of a set of *logically permissible transformations*, the rules shown in figure 5 on page 9. In the figure, by a slight abuse of notation, only the change to the tables is shown under the bar. The calculus guarantees that any sequence of applications of these rules gives a state that is a logical consequence of the initial state and the axioms used by the Reduction (2) rule.

3.2 Practical consequences

The data structures in memory manipulated by Factotum are a concrete representation of the *bind*, *reduce*, and *index* tables: *bind* is the memory, *reduce* is the set of indirection pointers, and *index* is the hash table used by the Sharer. We now describe the practical consequences of this choice of underlying semantics for the Factotum system.

The client code can build and replace nodes, thanks to the Construction and Reduction rules. These operations are performed using the higher-level interface described in section 2. Replacement is either explicit, following an indication by the client that two nodes are equivalent; or implicit, that is, induced by the modification of the label or the children associated with a node.

Factotum *systematically* uses the Collapsing, Rebinding, and Sharing I rules, to provide the most up-to-date information to the client code. By using these rules it thus makes improvements to the memory contents that persist after the client request.

The sharing subsystem *can* use the Add Index, Sharing II, and Indexing rules to improve sharing in the heap. Its strategy for applying these rules is completely internal, and transparent for the client.

Factotum therefore guarantees to the client that the result of its actions is correct for the equational semantics of the client operations.

4 Examples

To illustrate the ease with which the client programmer manipulates Factotum structures, we include in this section a short example of an operation on shared structures and the C++ code used to perform it. Other examples can be found in [She97a] and [She97b].

Consider the calculation of set difference in sharing trees[ZC95a]. Figure 6 shows an example of such a calculation. The intuition behind the algorithm is: traverse the first set, and try to follow in the second; as long as they agree, do not include anything in the result; at any point where they differ, copy the unsubtracted subtree from the minuend to the result, and go on to the next branch. (The "copy" should of course be shared with the original.) Using Factotum, we write the following code for set difference.

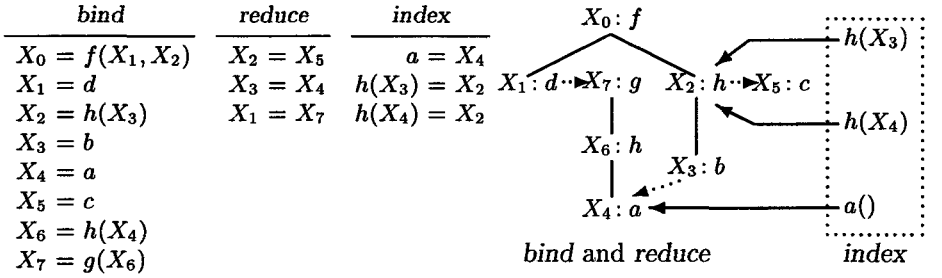


Fig. 4. Table contents representing $f(d, h(b))$ under rewriting. Solid lines are pointers, dotted lines are indirection pointers.

$$\text{Construction} \qquad \frac{X \text{ is not reachable from } X_0, \text{ nor from } \alpha}{bind[X \leftarrow \alpha]} \tag{1}$$

$$\text{Reduction} \qquad \frac{rterm\ X = \alpha, rterm\ Y = \beta, \Gamma \models \alpha \rightarrow \beta}{reduce[X \leftarrow Y]} \tag{2}$$

$$\text{Collapsing} \qquad \frac{reduce\ (reduce\ X) \neq \text{Nil}}{reduce[X \leftarrow reduce\ (reduce\ X)]} \tag{3}$$

$$\text{Rebinding} \qquad \frac{bind\ X = \alpha, reduce\ X = Y, bind\ Y = \beta \neq \text{Nil}}{bind[X \leftarrow \beta]} \tag{4}$$

$$\text{Sharing I} \qquad \frac{bind\ X = f(X_1, \ldots, X_i, \ldots, X_{\rho f}), reduce\ X_i = Y \neq \text{Nil}}{bind[X \leftarrow f(X_1, \ldots, Y, \ldots, X_{\rho f})]} \tag{5}$$

$$\text{Add Index} \qquad \frac{bind\ X = \alpha}{index[\alpha \leftarrow X]} \tag{6}$$

$$\text{Sharing II} \qquad \frac{bind\ X = bind\ Y \neq \text{Nil}}{reduce[X \leftarrow Y]} \tag{7}$$

$$\text{Indexing} \qquad \frac{Y = index\ (bind\ X) \neq \text{Nil}}{reduce[X \leftarrow Y]} \tag{8}$$

Fig. 5. Permissible transformations in the RWS calculus, where $X, Y \in \mathsf{Name}$, α and β are terms, and Γ is a set of axioms (rewriting rules).

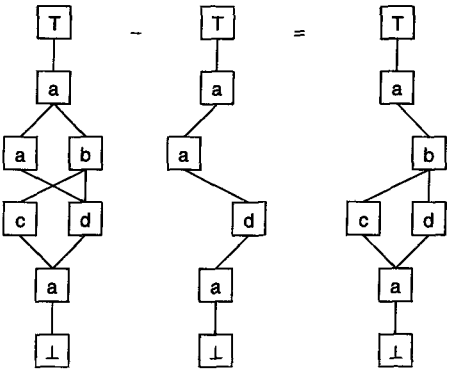


Fig. 6. Set difference in sharing trees

```
DFTraversal t(&minuend, &subtrahend, &result);
t.start();
while(!t.at_end())
  {
    if (t.status(&subtrahend) == valid)
      { // both agree so far; include nothing yet.
        t.next();
      }
    else
      { // disagreement; copy out unsubtracted part.
        if (t.status(&result) == invalid)
          { // result needs a path to here.
            add_spine(minuend, result);
          }
        result.replace(minuend.get());
        t.validate(&result);
        t.next_branch();
      }
  }
```

The auxiliary function `add_spine` copies the path from the root; once the spine is copied, we simply share the rest of the subtree from the minuend, and go on to the next branch of the subtrahend.

We claim that this code sufficiently high-level to be intuitive, and straightforward to validate. No explicit treatment of suffix sharing is necessary; it is handled by Factotum.

5 Experiments

The initial motivation for Factotum was the need to seamlessly integrate different representations of complex systems—n-ary decision diagrams,[5] sharing trees, functional evaluation—as part of the Clovis project. In this section we present some preliminary results of the associated implementation effort.

Our first experiments with Factotum have chiefly concerned the testing and validation of the higher-level programming interface. This approach permitted us to establish that the system as defined was useful, before investing in its in-depth optimization. Our experimental implementation uses the *cgc* lightweight generational garbage collector[cgc97] for memory management, and the fast dynamic hash tables of Larson[Lar88] as implemented by Strandh[Str92].

MEC[ABC94] is a model-checking program that computes the synchronization product of automata. These synchronization products can be computed by operations on explicit representations of automata, or using tuple-set operations [ZC95b]. In our first experiment we implemented a collection of tuple set operations inspired by Paquay's sharing trees implementation of MEC[Paq96]. An example of this experiment is the set difference example from section 4. A useful observation is that the systematic and transparent nature of sharing can be assumed in the application code, leading to more efficient algorithms. This can be seen in the example, where a difference between the two sets requires a copy of the unsubtracted subtree. Instead of a copy of this subtree, the line

```
result.replace(minuend.get());
```

shares the current node in `result` with the current subtree in `minuend`, taking constant time.

Toupie[RC93] is an interpreter of the propositional μ-calculus extended to finite symbolic domains, used to solve constraints and to describe properties of finite-state machines. In addition to the usual features of constraint languages, Toupie provides universal quantification and permits the definition of relations as fixpoints of equations. Its interest for this study is that its execution motor is based on n-ary decision diagrams, which critically need subtree sharing for efficiency, and memoization of function results.

The data structures and operations used for the Clovis project are directly inspired by Toupie. An important complication is that our variable domains are extended to arbitrary-precision integers and character strings. The former are necessary to adequately deal with nonbounded *counters* that arise in the modeling of systems described by VHDL specifications. The latter are simply convenient. In both cases, the liberty with which Factotum lets application programs define the labels on nodes greatly simplifies their coding. Rather than forcing us to inject the variable domains onto integer intervals, Factotum permits us to attach arbitrary labels, while retaining the performance advantages of direct access when such an injection is possible.

[5] Intuitively, BDDs extended to finite domains.

Clovis requires both decision diagrams for representing relations, and symbolic expressions extracted from VHDL programs or used for representing constraints. Both types of data structures are implemented using Factotum. The key challenges for a high-performance implementation of Clovis are, first, defining Toupie-like operators so that sharing is efficiently maintained; and second, making sure that the effects of memoization are obtained by proper sharing of common subexpressions.

A correct response to the first challenge is not a problem: using coupled traversals, we define standard operations for set union, intersection, and difference; constraint operations for join and selection (or cofactoring); and a parallel assignment used for stepwise simulation. Of particular practical interest is the tuning of the sharing strategy for overall efficient operation, a problem which is still under investigation.

The second challenge, concerning subterm sharing in symbolic expressions, resembles the use of sharing in term-rewriting to obtain the result of memoization that we described in chapter 6 of [She94]. The key idea seems to be that each projection of a relation on a set of variables be represented by an expression (subtree) that can be shared each time it appears, so the effect of its evaluation is available whenever the same projection appears in another expression. Work on these aspects continues.

6 Conclusions

The Factotum system provides an automatic and systematic sharing service for systems analysis applications that critically rely on sharing for good performance. It integrates congruence-based sharing techniques and high-level tools for manipulating structures in memory, and makes it possible to fine-tune sharing policy for overall efficiency. Existing sharing-aware applications, such as BDDs and tuple set operations using sharing trees, can clearly benefit from Factotum. But Factotum specifically aims at symbolic applications in general where validated automatic and systematic sharing provides a real advantage compared to explicit ad hoc hashing and memoization techniques.

The theoretical foundations of Factotum are provided by the RWS calculus, that gives an underlying equational semantics to the data structures in memory. This semantics is able, by design, to take into account shallow term equivalency and tabulation information stored in the data structures. Use of the RWS calculus lets Factotum guarantee that the results of its transformations, including persistent sharing improvements to the representation in memory, respect the equational semantics of the client operations.

Factotum provides a practical high-level programming interface for system implementors. `Name` are `Cursor` objects encapsulate fixed references (or roots) and arbitrary explorations below a fixed node, respectively. These objects protect the application code from address modifications induced by garbage collection and sharing, and provide a sophisticated programming interface suitable for mul-

tithreaded applications. Factotum also provides support for *coupled traversals*, which generalize a great many application-level operations.

Our experiments to-date have concentrated on validating the semantic model and programming interface, by reimplementation of the key parts of the existing tools MEC and Toupie. Further experiments are under way as part of the Clovis project.

A good deal of future work can already be foreseen. The next clear step is in-depth optimization of the Factotum code, based on back-to-back tests with existing systems. Experiments in related systems, such as the different approximations to congruence closure used in the *eqc* equational programming system, have shown that good results can be obtained with low-order constant overhead[She94,Mag94]; consequently we expect positive results from this engineering effort. A further step is the definition of adaptive strategies for the sharing subsystem, based on static or dynamic analysis of a given application, that evolve good performance without the hard-coding of application-specific strategies. This study must necessarily wait for the development of a body of Factotum examples.

Exploitation of the Factotum model in multithreaded and distributed applications is a further topic for research. The latter is of particular importance for analysis of industrial systems, where problem sizes are already beyond the capacity of monoprocessor machines.

References

[ABC94] André Arnold, Didier Bégay, and Paul Crubillé. *Construction and Analysis of Transition Systems with MEC.* Number 3 in AMAST Series in Computing. World Scientific Publishers, 1994.

[Ake78] B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[App87] Andrew Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4), 1987.

[Bry86] R. Bryant. Graph based algorithms for boolean fonction manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[cgc97] The CGC copying garbage collector. At ftp://ftp.labri.u-bordeaux.fr/, September 1997. Part of the *eqc* equational programming project.

[Che80] Leslie Paul Chew. An improved algorithm for computing with equations. In *21st Annual Symposium on Foundations of Computer Science*, 1980.

[HO82] Christoph Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, pages 68–95, 1982.

[Hug85] John Hughes. Lazy memo functions. In *Functional Programming Languages and Computer Architectures.* Springer-Verlag, 1985.

[Lar88] Per-Ake Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, April 1988.

[Mag94] Nicolas Magnier. Recalculs dans les systèmes de réécriture et programmation équationnelle. Technical Report 974-94, Laboratoire Bordelais de Recherche en Informatique, 1994.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):185–195, 1960.

[Mit68] D. Mitchie. 'Memo' functions and machine learning. *Nature*, pages 19–22, 1968.

[Paq96] Renaud Paquay. Implémentation du logiciel de vérification de modèle MEC avec les arbrés partagés. Master's Thesis, University Notre-Dame de la Paix, Namur, Belgium, 1996.

[Rau96] A. Rauzy. An introduction to binary decision diagrams and some of their applications to risk assessment. In O. Roux, editor, *Actes de l'école d'été, Modélisation et Vérification de Processus Parallèles, MOVEP'96*, 1996. Also Technical Report 1121-96, Laboratoire Bordelais de Recherche en Informatique.

[RC93] Antoine Rauzy and Marc-Michel Corsini. First experiments with Toupie. Technical Report 581-93, Laboratoire Bordelais de Recherche en Informatique, 1993.

[RV90] I. V. Ramakrishnan and R. Verma. Nonoblivious normalization algorithms for nonlinear systems. In *Proceedings of the International Conference on Automata, Languages, and Programming*, 1990.

[She90] David J. Sherman. Lazy directed congruence closure. Technical Report 90-028, University of Chicago Department of Computer Science, 1990.

[She94] David J. Sherman. *Run-time and Compile-time Improvements to Equational Programs.* PhD thesis, University of Chicago, Chicago, Illinois, 1994.

[She97a] David J. Sherman. Factotum: Automatic and systematic sharing support for symbolic computation. Technical Report 1174-97, Laboratoire Bordelais de Recherche en Informatique, September 1997.

[She97b] David J. Sherman. On referential transparency in the presence of uniform sharing. Technical Report 1179-97, Laboratoire Bordelais de Recherche en Informatique, October 1997.

[Str92] Robert Strandh. A dynamic hash library. Available at ftp://ftp.labri.u-bordeaux.fr/, March 1992. Based on Larson, CACM 31(4).

[Ver95] Rakesh M. Verma. A theory of using history for equational systems with applications. *Journal of the ACM*, 42(5):984–1020, 1995.

[Wil95] Paul R. Wilson. Garbage collection. *ACM Computing Surveys*, 1995. Available at file://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. Technical report, University of Texas, 1995. Available at file://ftp.cs.utexas.edu/pub/garbage/allocsurv.ps.

[ZC95a] D. Zampunieris and B. Le Charlier. Efficient handling of large sets of tuples with sharing trees. In *Proceedings of Data Compression Conference, DCC'95*, October 1995. Also Research Paper RP-94-004, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium.

[ZC95b] Didier Zampuniéris and Baudouin Le Charlier. An efficient algorithm to compute the synchronized product. In *Int'l Workshop MASCOTS*, 1995.