Program Derivation With Verified Transformations — A Case Study

J. P. KELLER
Kepler, Paris

AND

R. PAIGE
Courant Institute

Abstract

A program development methodology based on verified program transformations is described and illustrated through derivations of a high level bisimulation algorithm and an improved minimum-state DFA algorithm. Certain doubts that were raised about the correctness of an initial paper-and-pencil derivation of the DFA minimization algorithm were laid to rest by machine-checked formal proofs of the most difficult derivational steps. Although the protracted labor involved in designing and checking these proofs was almost overwhelming, the expense was somewhat offset by a successful reuse of major portions of these proofs. In particular, the DFA minimization algorithm is obtained by specializing and then extending the last step in the derivation of the high level bisimulation algorithm.

Our experience suggests that a major focus of future research should be aimed towards improving the technology of machine checkable proofs — their construction, presentation, and reuse. This paper demonstrates the importance of such a technology to the verification of programs and program transformations. We believe that the utility of transformational systems to program development will ultimately rest on a practical program correctness technology. © 1995 John Wiley & Sons, Inc.

1. Introduction

This paper describes a transformational programming methodology that includes a fully operational set-theoretic proof checker developed by the first author. The methodology is explained through illustrative examples of two moderately difficult program derivations. One of these derives a high level form of an algorithm to compute the bisimulation equivalence relation (see [51] and [60]) (which is used to model CCS program equivalence, [50], and also hyperset equivalence, [54]). The other derives an algorithm to minimize the number of states in a deterministic finite state automaton (DFA); see [52], [53], and [2]. Based on our experience with these derivations, we believe that mechanical verification of program transformations is the most important missing ingredient to the successful use of transformational programming as part of a viable program development technology.

1.1. Origins in SETL

The philosophical underpinnings of our work draw on two separate but related research directions pursued by Jack Schwartz and his collaborators within the

Communications on Pure and Applied Mathematics, Vol. XLVIII, 1–62 (1995)
© 1995 John Wiley & Sons, Inc. CCC 0010–3640/95/??0001–62

SETL project, which began almost thirty years ago. One direction had to do with the mechanization of a fully formal programming methodology powerful enough to derive the kind of combinatorial algorithms informally developed by Dijkstra in the second part of Discipline of Programming; see [27]. The other direction had to do with the extension of global program analysis techniques found in Cocke and Schwartz (see [18]) to support the compilation and optimization of a very high level programming language based on finite set theory.

The SETL language was central to both directions as a vehicle for demonstrating that set theory, which was accepted by the mathematics community as the lingua franca of mathematical abstraction, could support a fundamentally new, more productive and reliable software development technology. Schwartz proposed SETL (see [70]) as a language that would cost more to implement but, he believed, would make programming easier by providing a fixed but powerful collection of set-theoretic primitives that could model data structures and specify algorithms clearly and succinctly with a minimal amount of language extension. Development of the first validated Ada compiler (see [1]) implemented in SETL supported this view.

In order to place our own work in perspective, it is worth mentioning a few specific research contributions that came out of the SETL project. Schwartz was one of the earliest proponents of a comprehensive fully formal computer-aided system supporting program development by verified source program transformations based on a computational set theory. The programming language was SETL. The formal language and verification technology was based on Hoare logic (see [43]) with formulas consisting of SETL code fragments annotated with set-theoretic assertions; see [73]. Convenient algebraic rules for combining correct program fragments into larger correct fragments were a key feature. Davis and Schwartz developed a formal way to enable a mechanical verification system to extend itself with new proof methods without violating soundness and without changing the set of provable statements; see [24]. Edith Schonberg later extended Davis and Schwartz's verification framework to support the computer-assisted verification and application of program transformations; see [25]. She presented one of the most difficult transformational proofs in the literature — a formal derivation of Tarjan's graph reducibility algorithm (see [84]) — as a convincing case study. This early work inspired Cantone, Ferro, and Omodeo's work on fast decision procedures for fragments of set theory; see [13]. Early research on the generic program transformation method of set-theoretic finite differencing (an abstract mechanizable form of top-down stepwise refinement that was implemented by a generalized form of strength reduction, [18] and [17]) is found in [59], [58], and [75].

Implementation of an optimizing compiler was, of course, the greatest challenge of the SETL project, which spanned a period from the early 1970's through the 1980's. The two major goals for compiling SETL programs into efficient code were (1) to automatically select conventional storage structures for sets and maps and (2) to implement mathematical copy-value semantics without having to perform expensive copy operations. To achieve these goals, Schwartz and others in

the SETL project extended the methods of global program analysis discussed in his earlier work with Cocke (see [18]) to handle three new analysis problems: type inference, [86], analysis for set-theoretic inclusion and membership relations, [71], and analysis for multiple pointers to the same aggregate data, [72] and [69].

Much of the work on value flow and destructive update analysis is just now beginning to be digested by the functional programming and rapid prototyping communities. Tenenbaum's pioneering work in SETL type inference was especially successful, and has had wide influence. In particular, it had an impact on the European SED Esprit project directed by the first author. Within this project a programming environment for SETL, [30], was constructed within MENTOR, [31], using TYPOL, [26] and [45]; it implemented SETL type inference, [32], to facilitate SETL-to-Ada translation, [29].

Because none of the proposals for mechanizing a set-theoretic programming and verification methodology was actually implemented, the ideas were never put to a test. Although several unoptimized forms of SETL have been implemented (e.g., SETL2, [80], and CANTOR, [47], which was used to develop the NAP proof checker discussed in this paper), the optimizer was never fully operational (see [36], [76], and [82]), and some of the most basic theoretical ideas for SETL analysis and optimization were also never fully tested. In addition to a wealth of research results, the SETL project has left many interesting problems open for future investigation.

1.2. Transformational Programming

A program transformation is a mapping on well formed programs that preserves semantics. Transformational programming is a program development methodology whose origins can be traced back to Dijkstra's top-down stepwise refinement; see [28]. The basic idea is to map an initial formal problem specification into an implementation by successive applications of meaning-preserving transformations. A set of proven lemmas that each applied transformation preserves program semantics comprises a formal derivation that the final program implements the specification. The use of Hoare logic (see [43]) to prove that the specification satisfies input/output conditions may be added to the derivation to prove more convincingly that the implementation is "correct." When each abstract program transformed in this way can be expressed in a single language from the highest level specification down to the lowest level implementation, then Bauer called that language "wide spectrum"; see [3]. SETL was in this category. These rudimentary ideas evolved into a powerful programming methodology by Dijkstra, [27], and Gries, [40].

The top-down stepwise refinement approach promised to provide a more disciplined and goal-directed approach to program design in which derivational steps could be guided by complexity considerations to produce efficient programs. A top-down program derivation was a byproduct of construction that lent credibility to the proven code. It also offered to document deep semantic information about that code, which could facilitate redesign and maintenance. Key questions

remained about how hard it would be to design and prove transformations, and to recognize when they could be usefully applied.

However, any manual activity is prone to human factor difficulties. Key weaknesses of any purely programming methodology include the inconvenience of having to conform to a rigid discipline, the potential for human error in paper-and-pencil calculation, and the tedious, sometimes repetitive, low level, manual steps that are required. The point of a transformational methodology is to provide enough mechanization to make formal program development useful.

Two research directions have been taken towards the development of a useful semi-automatic transformational methodology. In one direction, exemplified by Burstall and Darlington (see [10]), the goal was to have a small, but complete (in the sense that any two equivalent programs can be transformed into each other) collection of program transformations. However, the difficulty of understanding how to verify Burstall and Darlington's Fold/Unfold transformations impeded mechanization. The total correctness of these transformations in the context of higher order functional languages was open for thirty years until Sands finally came up with a solution; see [68].

In the other approach, the goal was to have a large collection of easily mechanizable transformations specified as conditional rewriting rules; i.e., rewriting rules guaranteed to preserve correctness if conditions (limited to syntactically checkable program properties that implied semantic correctness) could be automatically verified. Cheatham and Wegbreit (see [15]) proposed this approach for use in the construction of experimental prototype optimizing compilers. Standish et al. later proposed a catalog of transformations of this kind; see [81]. Transformational systems developed with this approach made use of large transformational libraries of relatively low level transformations whose enabling predicates were proved by ad hoc simplifiers that relied mainly on local contextual information. These systems could group and sequence transformations to form simple automatic translators.

Transformational systems today, such as Kids, [79], and APTS, [57], are meta-systems in the sense of accepting formal syntax specifications. They also support global program analysis using attribute grammars or more sophisticated logic-based methods, and they incorporate conditional rewriting rules with conditions that are proved true based on the results of such global analysis. These systems facilitate the construction of powerful prototype systems for symbolic program analysis, manipulation, and translation.

However, these systems do not ensure soundness. Lacking from them (and sorely missed) is the mechanical support for verifying the correctness of transformations. The need to create confidence in the products of transformational systems rests upon preventing them from being used to derive invalid results. The need for machine checked verification of transformations has been expressed by Pepper in [62]. It is one of the most important and difficult goals of the Korso project (see [63]) and its KIV verification environment; see [64]. Currently it is also recognized to be highly labor intensive, and perhaps impractical for the most difficult theorems; see [66].

1.3. Motivation

Our choice of derivation came from two surprising experiences. We derived an improvement in Hopcroft's algorithm to compute a minimum state DFA (see [44]), which runs in asymptotically better space and matching worst case time. The most difficult step in the derivation involved application of a transformation whose correctness depended on a simply stated set-theoretic identity. A seemingly correct proof of this identity was worked out in mathematical style with paper and pencil. However, when the identity was then assigned to an honors graduate class in compilers at New York University, no student could prove it, and several students believed it was wrong. We found no error in the proof, but our confidence was shaken. Consequently, a formal mechanical proof was attempted using the NAP set-theoretic proof checker. After a great deal of work, there was an impasse, and a gap seemed to exist in the proof. We worked some more, filled the gap, and eventually obtained a mechanically checked proof.

The second surprise was that the Paige/Tarjan strong bisimulation algorithm was published with an error in the preprocessing phase; see [60]. The algorithm had been designed informally and without concern for an implementation, as is the style within the algorithms community. The error was minor and easily fixable, but we found out about it from scientists (outside the algorithms community) who wanted to implement it, and had difficulty finding a fix. We believe that this particular error would be almost impossible with a paper-and-pencil transformational derivation. Yet the error was introduced in the same portion of code obtained from application of a variant of the critical transformation that we proved for the DFA minimization derivation.

2. Mathematical Preliminaries

Our derivations will fit in the framework of Bloom and Paige (see [7]), where a paper-and-pencil transformational proof was given for a new improved ready simulation algorithm. That derivation made essential use of a SETL-like programming language, three basic transformations to map a fixed point specification into RAM code, and Hoare logic combined with set theory to verify transformations. However, the derivations to be presented here are somewhat more complex. More importantly, they make essential use of a new mechanical proof checking capability found in NAP (a set-theoretic proof checker designed and implemented by the first author) for verifying the most difficult transformations in our derivations.

2.1. Notation

The SETL-like notation (see [70], [74], and [80]) to be used throughout this paper is a concise variant of standard mathematical notation augmented with conventional dictions as may be found in an imperative programming language like Algol. Besides the usual elementary datatypes, SETL includes built-in datatypes for finite tuples (i.e., ordered collections of data) and finite sets. The elements of tuples and sets may be any datatype to an arbitrary finite depth of nesting.

Examples are the empty tuple (written [ ]), the empty set (written {}), and the 3-tuple $[\{\}, [true, [1], 2], \text{“}abs\text{”}]$ (whose first component is {}).

Set comprehension is, perhaps, the most useful expression in SETL. In its most general form it is written in the following way,

$$\{e(x_1, \ldots, x_k) : x_1 \in S_1, x_2 \in S_2(x_1), \ldots, x_k \in S_k(x_1, \ldots, x_{k-1}) | K(x_1, \ldots, x_k)\}$$

which denotes the set of element values $e(x_1, \ldots, x_k)$ evaluated over all k-tuples $[x_1, \ldots, x_k]$ belonging to the search space satisfying the condition,

$$x_1 \in S_1 \wedge x_2 \in S_2(x_1) \wedge \ldots \wedge x_k \in S_k(x_1, \ldots, x_{k-1})$$

and also satisfying the Boolean-valued qualifier $K(x_1, \ldots, x_k)$. Each variable x_i, $i = 1, \ldots, k$, that occurs in the iterator subpart (the text between symbols ‘:’ and ‘|’, which defines the search space) is bound to the set-valued expression $S_i(x_1, \ldots, x_{i-1})$. No applied occurrence of a bound variable of a set former can appear before its bound occurrence within the iterator, and all bound variables of a set former must be distinct.

When the qualifier subpart $K(x_1, \ldots, x_k)$ is the constant $true$, it may be elided. As a notational convenience, the frequently used set former

$$\{x : x \in S | K(x)\}$$

may be abbreviated

$$\{x \in S | K(x)\}$$

The bound variable convention of SETL set formers simplifies identification of bound and free variables. It also allows us to avoid writing existential quantifiers that would be required in conventional notation. For example, the union of two sets S and T may be expressed

$$\{y : x \in \{S, T\}, y \in x\}$$

rather than

$$\{y : \exists x \in \{S, T\} | y \in x\}$$

SETL maps are finite binary relations treated as multi-valued functions that are defined as sets of ordered pairs. Let R and R' be maps, and let S be a set. Then:

$$\begin{aligned}
\cup/S &= \bigcup_{x \in S} x \\
\mathsf{domain}\, R &= \{x : [x, y] \in R\} \\
\mathsf{range}\, R &= \{y : [x, y] \in R\} \\
R^{-1} &= \{[y, x] : [x, y] \in R\} \\
R\{x\} &= \{y : [u, y] \in R | u = x\} \\
R[S] &= \{y : [x, y] \in R | x \in S\} = \bigcup_{x \in S} R\{x\}
\end{aligned}$$

The preceding notation illustrates how expression $[x, y]$ can be used in a natural way within iterator $[x, y] \in R$ to bind variables x and y to the first and second components of pairs that belong to set R.

If S is a set, then $S\ with\ x$ and $S\ less\ x$ are convenient abbreviations for $S \cup \{x\}$ and $S - \{x\}$, respectively. The powerset of S is denoted by $pow(S)$. The cardinality of S is denoted by $\#S$. An arbitrary element can be selected from a set S by the expression $\ni S$.

SETL is an imperative language with assignment $a := b$ under copy/value semantics. If S is a set and *op* is any binary operator, then SETL allows assignment $S\ op := x$ to abbreviate $S := S\ op\ x$.

Image sets of maps can be updated by performing assignments of the following two kinds:

```
R{x} with:= y --element addition
R{x} less:= y --element deletion
```

or, equivalently, by writing,

```
R with:= [x,y]
R less:= [x,y]
```

When R is a many-to-one relation, we use the conventional notation for function application $R(a)$ to denote the unique b such that $[a, b] \in R$. $R(a)$ is undefined otherwise. In assignments,

```
R(a) := b
```

means

$$R := \{[a', b'] \in R | a' \neq a\}\, with [a, b]$$

which preserves the single-valued nature of R. SETL constant *om* denotes a special undefined value, and can be used to remove an element from the domain of a map, as in the example

```
R(a) := om
```

Finally, it is useful to follow the SETL (see [80] and [74]) convention of allowing Boolean-valued bounded existential quantifiers $\exists x \in S | K$ to have a side effect assignment. Whenever the quantifier evaluates to true, then variable x is assigned an arbitrary value that belongs to S and to the truth set of qualifier K. This convention allows us to use such existential quantifiers as the condition of a while-loop whose body can refer to x in order to retrieve the truth set value newly chosen before each while-loop iteration.

SETL contains a special looping construct,

```
for x ∈ S loop
    <a sequence of statements goes here>
end loop
```

which searches through a set S without repetition, and assigns successive values of S to x, which can then be used within the body of the loop.

SETL's built-in set datatype supports two distinctive and powerful features — nondeterminism and associative access — which makes SETL programming easier than many other languages, even when SETL is restricted to its most primitive set operations. Both the for-loop and arbitrary selection support nondeterministic programming, which relieves the programmer from having to choose an implementation for a particular search strategy. SETL's membership test $x \in S$ and map application $f(x)$ are primitive associative access operations that test whether a value belongs to a set. In contrast to conventional languages, which encourage the storage and retrieval of data indirectly through location expressions (pointers or array indexes), SETL encourages the direct use of essential data to access and navigate through other *related* data. Navigational paths are consequently short; programs are concrete and readable. A single SETL primitive for element deletion might require a dozen statements in C.

2.2. Derivational Tools

Before proceeding with the derivation, we first need to describe three generic transformational tools that can be used to map an abstract problem specification into efficient RAM code.

Dominated Convergence

A partially ordered set $(X, \leqq)$ is a set X with a reflexive, anti-symmetric, transitive relation $\leqq$. It has the *descending chain condition* if every totally ordered subset of X that includes a maximum element is finite. Given a function $f : X \to X$, we define $f^0(x) = x$ and $f^{n+1}(x) = f(f^n(x))$, and use the term $\text{gfp}\, x.f(x)$ to denote the greatest fixed point of f. Function f is said to be *deflationary* if $f(a) \leqq a$ for every $a \in X$. f is *monotone* if $f(a) \leqq f(b)$ for every $a, b \in X$ such that $a \leqq b$.

We will use the fixed point theorem and dominated convergence argument just below as tools for proving correctness of a formal specification, and for taking the initial step in the algorithm derivation.

DEFINITION 2.1. *A semilattice is a structure $(L, \sqcap, 1)$, where L is a set, $\sqcap$ is a binary meet (i.e., greatest lower bound) operation, and 1 is the greatest element in the partial order induced by $\sqcap$: $x \leqq y$ if and only if $x = x \sqcap y$.*

THEOREM 2.2. (TARSKI — [85]) *Given a semilattice $(L, \sqcap, 1)$ which has the descending chain condition, and given a monotone function $f : L \to L$, then*

$$(\max Y | Y \leqq f(Y)) = \text{gfp}\, Y.f(Y) = f^i(1)$$

for some finite i.

THEOREM 2.3. (DOMINATED CONVERGENCE — [12]) *Under the same conditions as in Theorem 2.2, and given any sequence s_i such that*

1. $s_0 = 1$
2. $s_{i+1} = s_i$ *if* $s_i = f(s_i)$
3. $s_i > s_{i+1} \geqq f(s_i)$ *otherwise.*

then s_i converges to $s_k = \mathrm{gfp}\, Y.f(Y)$ for some finite k.

The idea of dominated convergence is to compute a sequence $[s_i : i = 0, 1, \ldots]$ that converges to the same fixed point as the Tarski sequence $\left[f^i(1) : i = 0, 1, \ldots\right]$, and satisfies $s_i \geqq f^i(1)$ for $i = 0, 1, \ldots$. We seek a dominated sequence that takes more steps to reach the fixed point but at less overall cost than the Tarski sequence. Dominated convergence can be used to compute fixed points over a variety of lattice-theoretically defined datatypes. In [7] the ready simulation problem was solved by using dominated convergence to compute the fixed point of a set-valued function defined on a powerset lattice.

If S is a finite nonempty set, then P is a *partition* of S if and only if P is a set of nonempty mutually disjoint subsets of S whose union is all of S. The two problems considered in this paper can be specified as fixed points of functions defined on partitions. Elements of a partition are sometimes called *blocks*. Let $\mathcal{P}(S)$ denote the set of all partitions over S. If $Q, P \in \mathcal{P}(S)$, then we say that Q is a *refinement* of P, denoted by $Q \leqq P$, if and only if $\forall q \in Q | (\exists p \in P | q \subseteq p)$. The partially ordered set $(\mathcal{P}(S), \leqq)$ has maximum element $\{S\}$ and minimum element $\{\{x\} : x \in S\}$, and, being finite, it must have the descending chain condition.

Consider the following basic operation on partitions:

DEFINITION 2.4. *If P is a partition of a finite set S, and $a \subseteq S$ (i.e., $a \in pow(S)$). Then*

$$P \; split \; a \stackrel{\text{def}}{=} \{s \in pow(S) | s \neq \{\} \wedge (\exists p \in P | s \in \{p \cap a, p - a\})\}$$

Basic properties of split, which follow directly from the definition, are summarized below.

PROPOSITION 2.5. *Let P and Q be two partitions of finite set S, and let $a, b \subseteq S$.*

1. *P split $a \leqq P$ (split is deflationary in its first argument)*
2. *if $P \leqq Q$, then P split $a \leqq Q$ split a (split is monotone in its first argument)*
3. *P split $a = P$ if and only if $\exists A \subseteq P | a = \cup / A$ (splitting a partition has no effect if and only if one splits by any union of elements of P)*
4. *P split $a = P$ if and only if $\forall p \in P | p \subseteq a \vee p \cap a = \{\}$*

5. $a = \cup/\{x \in P \; split \; a | x \subseteq a\}$ *(Set a is exactly covered by a subset of P split a)*
6. *If* $p \in P$, $a \subseteq p$, $a \neq p$, *and* $a \neq \{\}$, *then* $P \; split \; a = (P - \{p\}) \cup \{a, p - a\}$ *(Splitting P by a set contained within some block $p \in P$ is a local change in which p is replaced by a and $p - a$.)*
7. *If* $A \subseteq P$, $A \neq \{\}$, $a = \cup/A$, $b \subset a$, *and* $b \neq a$, *then* $P \; split \; b = (P - A) \cup A \; split \; b$ *(Splitting by a small set affects only the blocks that intersect that set.)*
8. $(P \; split \; a) \; split \; b = (P \; split \; b) \; split \; a$ *(The result of splitting with respect to more than one set is the same regardless of the splitting order.)*

Based on the last property of Proposition 2.5, we can form a collective split operation that splits a partition relative to a collection of sets. It is also useful to treat such collections formally, as was done for partitions. If A and B are sets of subsets of a finite set S, then $B \sqsubseteq A$ if and only if $\forall a \in A, \exists D \subseteq B | \bigcup / D = a$. The space $(pow(pow(S)), \sqsubseteq)$ forms a partially ordered set with maximum element $\{\}$ and minimum element $\{\{x\} : x \in S\}$.

DEFINITION 2.6. *Let P be a partition over a finite set S, let A be a collection of subsets of S, and let a be any subset of S. Then we have the following recursive function definition.*

$$\begin{aligned} P \; split\% \; \{\} &= P \\ P \; split\% \; A \; with \; a &= (P \; split \; a) \; split\% \; A \end{aligned}$$

Properties of *split*% that follow directly from its definition are summarized below.

PROPOSITION 2.7. *Let Q and P be partitions over a finite nonempty set S, and let A, B, and R be subsets of $pow(S)$.*

1. *If* $Q \leqq P$ *and* $A \sqsubseteq B$, *then* $Q \; split\% \; A \leqq P \; split\% \; B$. *(split% is monotone in both arguments.)*
2. $P \; split\% \; B \leqq P$. *(split% is deflationary in its first argument.)*
3. $P \; split\% \; A = P$ *if and only if* $\forall p \in P, \forall a \in A | p \subseteq a \vee p \cap a = \{\}$ *(A is a right identity)*
4. $P \; split\% \; (A \cup B) = (P \; split\% \; A) \; split\% \; B$
5. $P = \{S\} \; split\% \; P$
6. $P \; split\% \; P = P$ *(idempotence).*
7. $P \; split\% \; Q = Q \; split\% \; P$ *(commutativity).*
8. $(P \; split\% \; Q) \; split\% \; R = P \; split\% \; (Q \; split\% \; R)$ *(associativity).*

9. *If $a \subseteq S$, P split $a = P$, R partitions a, and $b \in R$, then P split% (R less b) = P split% R (A crucial property used in the partitioning algorithms of Cardon and Crochemore; see [14]).*

10. *The meet operation on partitions is defined by* $Q \sqcap P \stackrel{\text{def}}{=} Q$ *split%* P.

A partition P is said to be *stable* with respect to set a if and only if the conditions of Proposition 2.5 (4) hold. P is stable relative to a collection A of sets if the conditions of Proposition 2.7 (3) hold. If S is a finite nonempty set, then $(L = \mathcal{P}(S), split\%, \{S\})$ forms a semilattice by Proposition 2.7 (10). Since L has a descending chain condition, we can exploit the Dominated Convergence Theorem to compute fixed points of monotone functions defined on L.

COROLLARY 2.8. *Let S be a finite set with $L = \mathcal{P}(S)$, and let $f : L \to L$ be a monotone, computable function. Then* gfp *Y.f(Y) can be computed in $O(\#S)$ iterations by the following code:*

```
-- maintain invariant f(Y) ≦ Y
Y := {S};
while ∃ x ∈ (f(Y) − Y) loop
    -- execution frequency O(#S)
    Y := Y split x
end loop
```

Proof: The preceding code assigns successive values s_i to Y, where $s_0 = \{S\}$, $s_{i+1} = s_i$ if $s_i = f(s_i)$, and $s_{i+1} \in \{s_i \; split \; x : x \in f(s_i) - s_i\}$ otherwise. It is shown by induction on i that $s_{i+1} < s_i$ whenever $s_i \neq f(s_i)$, and $f^{i+1}(\{S\}) \leqq f(s_i) \leqq s_{i+1} \leqq s_i \leqq \{S\}$ for all i.

(Basis) If $s_0 = \{S\}$ is a fixed point of f, then all of the inequalities become equalities, and the claim holds. Otherwise, $\{S\} > f(\{S\})$, since $\{S\}$ is the maximum partition. Hence, there must be some $x \in (f(\{S\}) - \{S\})$. Let $s_1 = \{S\} \; split \; x$. By Proposition 2.5 (3), $\{S\} \; split \; x < \{S\}$ and $f(\{S\}) \; split \; x = f(\{S\})$. By monotonicity of *split*, $f(\{S\}) \; split \; x \leqq \{S\} \; split \; x$, and the claim holds.

(Induction) Assume the claim holds for any $i > 0$. Then $f^{i+1}(\{S\}) \leqq s_{i+1} \leqq s_i$, and $f(s_i) \leqq s_{i+1}$. By monotonicity of f, we have $f^{i+2}(\{S\}) \leqq f(s_{i+1}) \leqq f(s_i)$, which implies that $f^{i+2}(\{S\}) \leqq f(s_{i+1}) \leqq s_{i+1}$. If s_{i+1} is a fixed point, then $f(s_{i+1}) = s_{i+2} = s_{i+1}$, and the claim holds. Otherwise, $f(s_{i+1}) < s_{i+1}$, and there must be some $x \in f(s_{i+1}) - s_{i+1}$. Let $s_{i+2} = s_{i+1} \; split \; x$. By Proposition 2.5 (3), $s_{i+1} \; split \; x < s_i$, and $f(s_{i+1}) \; split \; x = f(s_{i+1})$. By monotonicity of *split*, $f(s_{i+1}) \; split \; x \leqq s_{i+2} = s_{i+1} \; split \; x$. The result follows.

Finite Differencing

Whereas dominated convergence computes fixed points according to a rudimentary strategy, the finite differencing transformation (see [58]) maintains and exploits program invariants that are used to implement this strategy efficiently. The idea underlying this transformation has been called "strengthening invariants" by Dijkstra in [27] and Gries in [40]. Related techniques go back to the compiler optimizations for strength reduction, [18], [17], iterator inversion, [33], and set-theoretic strength reduction, [35], [59].

The essential idea is to avoid costly repeated calculations $f(x_1, \ldots, x_k)$ occurring in a program region R by maintaining equality invariants of the form

$$C = f(x_1, \ldots, x_k) \tag{2.1}$$

where $C, x_1, \ldots, x_k$ are distinct variables. The finite difference transformation inserts code on entry to R that establishes invariant (2.1). Within R just before each assignment $x_i := g(x_i)$ to a variable on which f depends spoils invariant (2.1), code is inserted to update variable C in order to re-establish invariant (2.1) immediately after the assignment to x_i. All calculations of $f(x_1, \ldots, x_k)$ that are made redundant by these steps are replaced by the variable C. Finite differencing improves performance when the cumulative cost of repeatedly computing $f(x_1, \ldots, x_k)$ in the unoptimized program exceeds the cost of establishing and maintaining invariant (2.1) in the optimized program.

Finite differencing rules can be formalized using Hoare Logic.

DEFINITION 2.9. *Let S be a command in an imperative programming language, and let φ and φ' be assertions in a formal language, where the free variables of φ and φ' are also program variables. The Hoare formula $\{\varphi\}S\{\varphi'\}$ is said to be satisfied if and only if every terminating execution of S that satisfies φ in its initial state satisfies φ' in its final state.*

The following lemma, which goes back to the general strength reduction framework of Cocke and Schwartz in [18], and was used throughout the derivation in [7], provides a basic schema for designing correct code to maintain equality invariants of the form (2.1). This code is made more efficient by straightforward simplification.

LEMMA 2.10. (FINITE DIFFERENCE SCHEMA) *Let f, g, and h be computable functions, let $C, x_1, x_2, \ldots, x_k$ be distinct program variables, and let φ represent formula*

$$f(x_1, \ldots, x_{i-1}, g(x_i), x_{i+1}, \ldots, x_k) = h(f(x_1, \ldots, x_k), x_1, \ldots, x_k) \tag{2.2}$$

which we call a differential identity for f. Then the following Hoare formula is satisfied:

$$\{\varphi \wedge C = f(x_1, \ldots, x_k)\}$$
$$C := h(C, x_1, \ldots, x_k)$$
$$x_i := g(x_i)$$
$$\{C = f(x_1, \ldots, x_k)\}$$

Real-Time Simulation

Finite differencing supports efficient navigation using low level SETL primitives. However, we still need an efficient method of implementing these primitives — especially associative access operations such as membership testing and map application. For this purpose we use a data structure design tool that generalizes Schwartz's method of basings; see [71].

Basings was one of the most ingenious ideas to come out of the SETL project. It was a way to aggregate data (related by storage or retrieval operations) around finite sets called "bases" that were used like a "bulletin board." For example, suppose that all the elements of two sets A and B were stored in the same base. Suppose also that both A and B were implemented as linked lists of pointers to their element values stored in the base. Then we could compute the union of A and B, and store the result in a third set C in linear time, regardless of the sizes of the elements. Just traverse one of the lists, say A, and mark each member of the base that belongs to A. At the same time, C is constructed as a linked list of pointers to marked elements of the base. Then traverse the other list B, and extend C only when we see an unmarked element of the base.

The SETL implementation fell short of this sort of efficiency in favor of hashed data structures that could support a fast general purpose read method (to detect and eliminate duplicate elements of deeply nested sets), and could respond well to unpredictable worst case situations. In SETL, sets A, B, and the base would be implemented as linked hash tables. A new hash table for C would also be constructed if it had to be searched (as for printing). Otherwise, C could be represented by marked elements in the base. As in our implementation, A and B would be searched efficiently using their linked list implementations. Although the "bulletin board" approach was clearly present in Schwartz's technique, the persistence of hashing in the best data structures available within the runtime system made the performance of SETL programs unpredictable.

In [55], the bulletin board idea underlying Basings was formalized and extended into a generic technique for arrays and linked lists. Sufficient conditions were given for a method that could simulate SETL-like associative access $x \in S$ on a RAM in real-time; i.e., access could be performed in unit worst case time and space on a RAM independent of the size of the data. Unfortunately, no efficient read method was given. That came later in [56].

The real-time simulation method just mentioned only works on a highly restrictive class of SETL2 programs that read all of their input at the beginning

of execution. A simplified form of the method is sketched here. More details are found in [11].

Assume that each distinct program variable v is restricted to values that belong to a set denoted by some type τ throughout the execution of the program. Such a program invariant is specified by a type assignment $v : \tau$. Consider the following simple type system, where each type τ represents a set $val(\tau)$. Type INT stands for the set of integers. If τ is a type, then type $set(\tau)$ represents the set of finite subsets of $val(\tau)$. If τ_1 and τ_2 are types, then $smap(\tau_1, \tau_2)$ and $mmap(\tau_1, \tau_2)$ represent the set of finite partial single-valued (resp. multi-valued) maps from domain $val(\tau_1)$ to range $val(\tau_2)$. Type variable B is called a base type, and is uniquely associated with a subtype constraint of the form $B < \tau$ (where τ is any type except for a base type, although τ may contain occurrences of base types), which indicates that $val(B)$ is a subset of $val(\tau)$. Every base type B must be associated with a single subtype constraint of the form $B < \tau_1$. The subtype constraints for a given program are restricted so that the graph

$$\{[B_1, B_2] : B_1 < \tau_2 \textit{ is a subtype constraint and base type } B_2 \textit{ occurs in } \tau_2\}$$

is acyclic.

Besides representing specific sets, each type is also used to model specific data structures needed to implement set and map operations efficiently. For this purpose, we augment the type system slightly by allowing a base type B to appear in two forms, called weak and strong, and denoted by B and $B - strong$ respectively. Three kinds of data structures are considered — *unbased*, *weakly based*, and *strongly based*. An unbased set is a set whose element type is not a base type. It is implemented as a doubly linked list of element values. An unbased map is a map whose domain element type is not a base type. Its domain is implemented as a linked list. Each image set is stored separately, and is accessible from its corresponding domain value by pointer.

Each base type B with subtype constraint $B < \tau_1$ is implemented as a set R_B of records, whose key field stores all the values in $val(B)$; i.e., the key field of each record stores a distinct value of type τ_1 that belongs to $val(B)$. Weakly based sets, denoted by $set(B)$, are implemented by doubly linked lists of pointers to records in R_B whose key fields store the elements of these sets (see Figure 1). Weakly based maps, denoted by $smap(B, \tau_1)$ and $mmap(B, \tau_1)$, have similar linked list implementations for their domains. Strongly based sets are denoted by $set(B - strong)$; strongly based maps are denoted by $smap(B - strong, \tau_1)$ and $mmap(B - strong, \tau_1)$. The records of R_B have a distinct field for each set or map that is strongly based on B. If S is a strongly based set, its corresponding field is used to store a doubly linked list that connects the set of records in R_B whose key fields store elements of S. The domains of strongly based maps are implemented in a similar way (see Figure 2). This representation supports unit-time element deletion and linear time search.

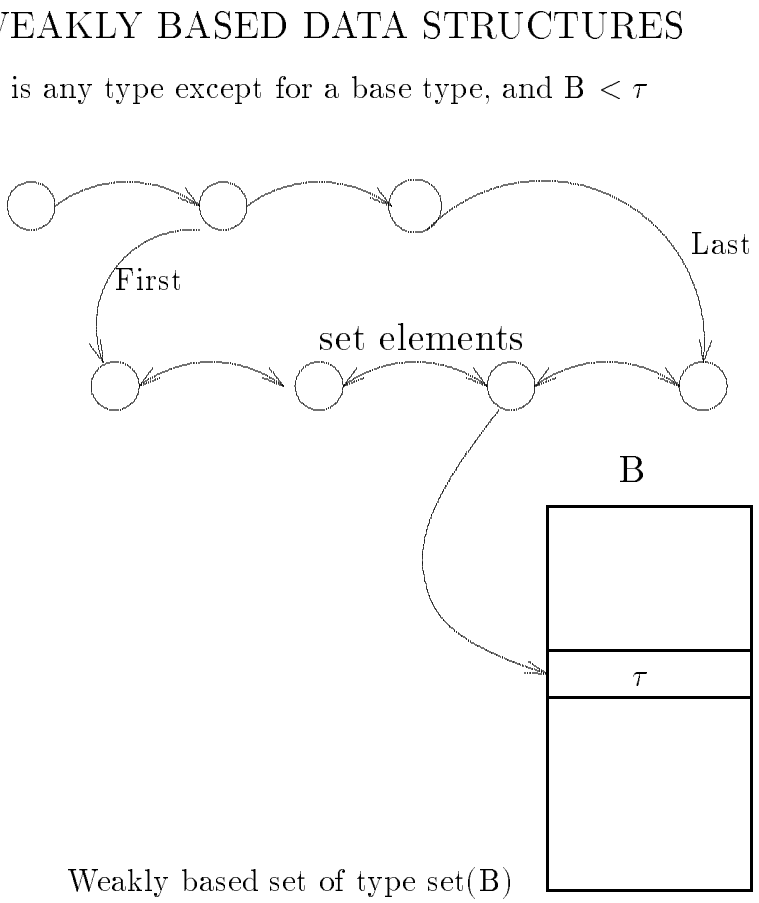

Figure 1. Weakly based data structures.

For each base type B, $val(B)$ is defined to be the smallest set that satisfies the program type assignments (restricted to input variables and constants) and the program subtype constraints. In [11] it is shown how to calculate $val(B)$ for any base type B as a finite set of values obtained from the input and from program constants.

The preceding types and corresponding data structures can be used to analyze the asymptotic time and space bounds utilized by SETL programs formally. Consider the type assignment $v : B$, $y : C$, $g : smap(B - strong, \tau')$, $f : mmap(B - strong, C)$, $T : set(C - strong)$, $R : set(C)$, and $S : set(B - strong)$, with subtype constraints $B < \tau$ and $C < \tau'$. Then operations $v \in S$, $S\ less := v$, $S\ with := v$, $g(v) := y$ can all be done in unit time.

Assignment $T := f[S]$ can be performed in time $O(\#S + \#\{[x, y] \in f | x \in S\})$, as can be seen from the following implementation:

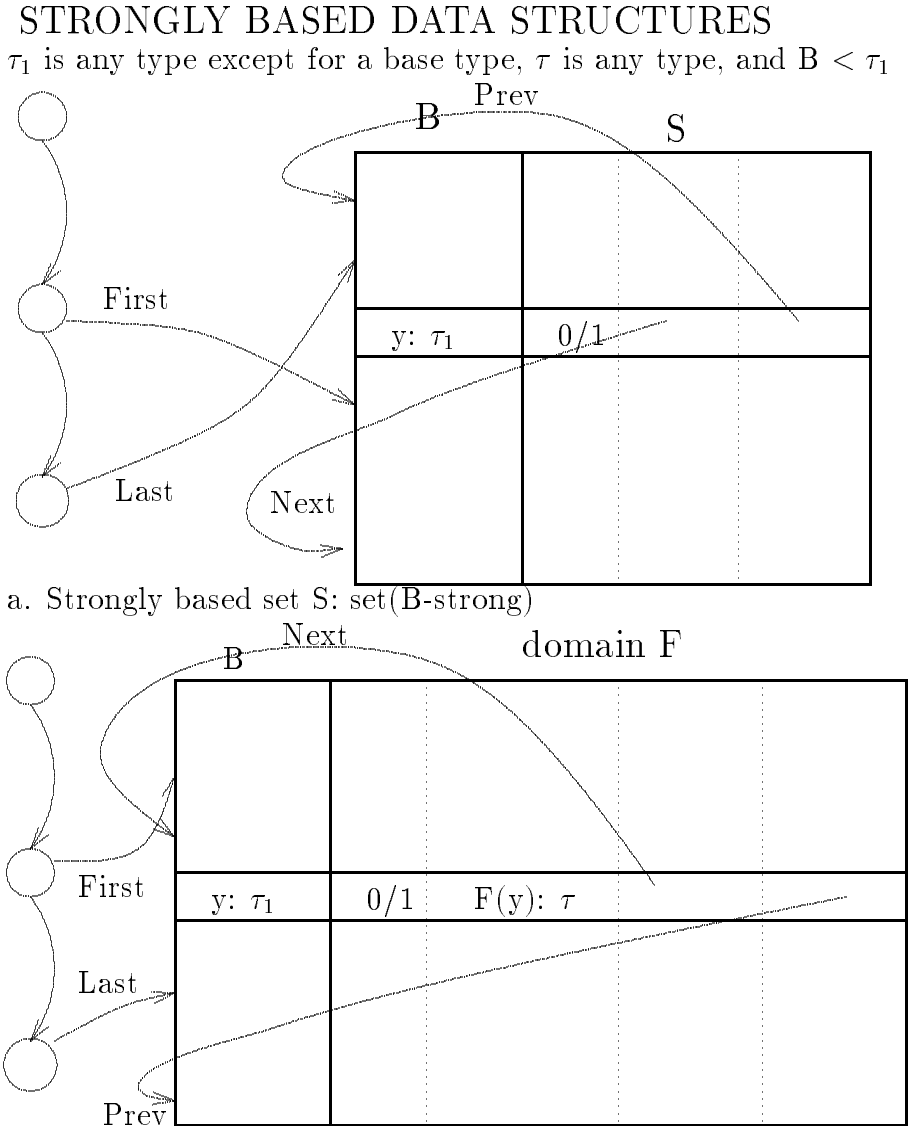

Figure 2. Strongly based data structures.

```
T := {}
for x ∈ S loop
    -- execution frequency #S
    for y ∈ f{x} loop
        -- execution frequency #{[x,y] ∈ f|x ∈ S}
        if y ∉ T then
            T with:= y
        end if
    end loop
end loop
```

where iteration through a set takes linear time in the cardinality of the set, and each of the other operations takes unit time. Given type assignment $T' : set(C)$, we can split set T into two sets $T - R$ and $T \cap R$ in time $O(\#R)$ by destructive modification to T,

```
T := T − R
T' := T ∩ R
```

using the following lower level implementation:

```
T' := {}
for y ∈ R loop
    -- execution frequency #R
    if y ∈ T then
        T less:= y
        T' with:= y
    end if
end loop
```

All of the time bounds just given hold for any types τ and τ' other than base types. In contrast, under the most optimistic assumptions for a hash table implementation, the expected cost of computing a single primitive associative access, say $f(x)$, would grow linearly in the size of x for the case where $x \in$ domain f.

Our approach to algorithm derivation depends strongly on guidance from complexity considerations. Hence, before discussing the derivation we need to carry out a thorough analysis of the essential datatypes and operations used in the problem domain. The real-time simulation technique makes such an analysis possible with minimal involvement in low level implementation detail.

The main datatype for our problem domain is the partition semilattice together with the *split* operation. For an efficient implementation we need to augment our type system with two new features. Whenever the element types of the domain and range of a single valued map are both base types, say B and C, then there is a convenient data structure to implement type $smap(B - strong, C - strong)$, which is strongly based in its range as well as its domain. Since the preimages of such a map are pair-wise disjoint, each preimage can be

separately linked, and all of the preimages can be stored in a single column. The range can be stored like a strongly based set, where each range element points to some one arbitrary witness in its preimage. If map f is of this type, and we also have $x : B$ and $y : C$, then single-valued map operations $f(x) := om$ (which removes x from a unique preimage set) and $f(x) := y$ (which adds x to preimage set $f^{-1}\{y\}$) can be performed in unit time. Preimage set $f^{-1}\{y\}$ can also be accessed directly from range element y in unit time; it can be subsequently searched in linear time with respect to its cardinality.

We will also want to have operation $new(B)$ to create a new value of type B dynamically. Although we will not need it in this paper, $delete(x)$ is an obviously useful thing to have for explicit garbage collection of unused base elements.

These new features allow us to represent partitions that can be updated easily in situ. Partitions that are represented as a set of sets are harder to modify in this way, and require closer scrutiny to ensure that they remain partitions after they are modified.

Suppose that partition P over set S is input as a set of sets, where the elements of S are represented by integers. More formally, suppose we have type assignment $P : set(set(node))$, where elements of each block of P are of type $node < INT$. Using the multiset set discrimination procedure found in [56] (based on [60]), an input method can remove duplicate blocks and ensure that the remaining blocks of P are mutually disjoint in linear time with respect to the length of the input string.

Consider how to convert P into the alternative representation $T1 : smap\ (node - strong, thin - strong)$, where $thin < INT$ is a base type used to "name" blocks (preimage sets) in $T1$, and where

$$P = \{T1^{-1}\{q\} : q \in \mathsf{range}\, T1\} \tag{2.3}$$

Let smap $pname = \{[p, new(thin)] : p \in P\}$ "name" each block of P uniquely. Then the conversion can be done in time $O(\#S)$ by implementing assignment

$T1$:= $\{[x, pname(p)] : p \in P, x \in p\}$

using the following lower level code (based on the operational semantics of SETL set formers)

```
for p ∈ P loop
    id := new(thin) -- id = pname(p)
    for x ∈ p loop
        -- execution frequency #S
        -- place all elements of block p in the same preimage of
        T1 T1(x) := id
    end loop
end loop
```

We can also split $T1$ relative to any set $splitter \subseteq S$, in a way corresponding to $P\ split\ splitter$, so that relationship (2.3) is preserved. Define two smaps

$T1'$ and $T1''$ to be equivalent, written $T1' \equiv T1''$, if their respective sets of preimages form the same partition. Let smap $pname = \{[p, new(thin)] : p \in P\ split\ splitter\}$ "name" each block of $P\ split\ splitter$ uniquely. Let $twin : smap(thin - strong, thin)$ map "names" of blocks in $T1$ that intersect with $splitter : set(node)$ into new "names"; i.e.,

$$twin = \{[n, new(thin)] : n \in T1[splitter]\}$$

Then the following equivalences,

$$\begin{aligned} & \{[x, pname(p)] : p \in P\ split\ splitter, x \in p\} \\ \equiv\ & \{[x, twin(n)] : n \in T1[splitter], x \in splitter \cap T1^{-1}\{n\}\} \cup \\ & \{[x, y] \in T1 | x \notin splitter\} \\ \equiv\ & \{[x, twin(T1(x))] : x \in splitter\} \cup \{[x, y] \in T1 | x \notin splitter\} \end{aligned}$$

ensure that $T1$ can be split by $splitter$ using assignments,

$$\begin{aligned} twin &:= \{[n, new(thin)] : n \in T1[splitter]\} \\ T1 &:= T1 - \{[x, y] \in T1 | x \in splitter\} \cup \{[x, twin(T1(x))] : x \in splitter\} \end{aligned}$$

so that invariant (2.3) is preserved. The procedure just below is a lower level implementation, which takes time and space $O(\#splitter)$:

```
-- call by reference
procedure split(T1, splitter)
twin := {}
for x ∈ splitter loop
    -- execution frequency #splitter
    qold := T1(x)
    qnew := twin(qold)
    if qnew = om then
        -- generate a new base element that 'names' the new block
        qnew := new(thin)
        twin(qold) := qnew
    end if
    -- indexed assignment to T1 is broken into two operations
    T1(x) := om      --remove x from block T1^-1{qold}
    -- we could execute delete(qold) if T1^-1{qold} = {}
    T1(x) := qnew    --add x to block T1^-1{qnew}
end loop
end
```

The data structure for *node* can be represented by either an array or by an unconnected set of records. However, since base *thin* is dynamic, we prefer an implementation by separate records — not by an array.

Suppose that we repeatedly call the preceding *split* procedure to execute $T1\ split\ f^{-1}[q]$, where f^{-1} is drawn from a family of inverse maps $\mathcal{F}'$. Suppose also that we have the type assignment

$\mathcal{F}' : set(mmap(node - strong, node))$

where an array is used to implement $node$, and each strongly based map $f^{-1} \in \mathcal{F}'$ is implemented as an array-based adjacency list (either as a column of the base data structure or a detached vector). Then each preimage set calculation $f^{-1}[q]$ to form a splitter is achieved in time $O(\#q + \#\{[y,x] \in f^{-1} | y \in q\})$, and *split* is performed in time $O(\#f^{-1}[q])$, as reported above. These data structures are similar to those used in Hopcroft's DFA minimization algorithm; see [44].

3. Derivation

Having gained a basic understanding of partitions and their costs, we can begin the derivation. First, we will derive a high level, generalized form of the bisimulation algorithm of Paige and Tarjan; see [60]. This is sufficient to correct a minor error found in that paper. Next, we will transform the bisimulation algorithm into an improved form of Hopcroft's algorithm to minimize a deterministic finite state automaton; see [44]. The derivations will be short, will reveal the essential algorithmic decisions, and will facilitate a complexity argument. In this section the most difficult transformational step, the one that has been proof-checked mechanically, will be accepted on faith. The next section will describe the formal proof in detail.

Previously Fernandez in [34] and Westin in [89] gave partial derivations of a generalized form of the fast bisimulation algorithm found in [60]. Fernandez's algorithm was not quite the same, and Westin's algorithm did not achieve the best complexity, because he was unable to fix the error that appeared in [60]. In [39] Gries gave a full derivation of Hopcroft's algorithm, but that derivation was lengthy, and did not capture the essential algorithmic decisions that Hopcroft made. In [12] Cai and Paige later presented the initial portion of a shortened derivation similar to the one given here, and they speculated that the full derivation would yield an algorithm that improves Hopcroft's algorithm in space utilization. In [87] Watson also derived the Hopcroft algorithm in a more intuitive way than did Gries. However, his derivation was not intuitive enough to support an easy and formal algorithmic analysis of the resulting algorithm.

3.1. Specification

Like Smolka and Kanellakis (see [46]) we will model the bisimulation problem mathematically as the *relational coarsest partition problem*. Given a partition P of a finite set S, and a nonempty finite family $\mathcal{F}$ of binary relations over S, compute the maximum refinement Q of P such that Q is stable relative to every preimage $f^{-1}[q]$ for every $f \in \mathcal{F}$ and every $q \in Q$; i.e.,

$$\max\ Q \leqq P | (\forall q \in Q, \forall f \in \mathcal{F}, \forall r \in Q | q \subseteq f^{-1}[r] \vee q \cap f^{-1}[r] = \{\}) \tag{3.1}$$

After rewriting specification (3.1) equivalently as

$$\max\ Q \leqq P | (\forall q \in Q, \forall b \in \{f^{-1}[r] : f \in \mathcal{F}, r \in Q\} | q \subseteq b \vee q \cap b = \{\})$$

we can invoke Proposition 2.7 (3) in order to place the specification directly into the more convenient form,

$$\max\ Q \leqq P | Q = Q\ split\% \left\{ f^{-1}[r] : f \in \mathcal{F}, r \in Q \right\}) \tag{3.2}$$

Syntactic analysis of specification (3.2) tells us that subexpression

$$\left\{ f^{-1}[r] : f \in \mathcal{F}, r \in Q \right\}$$

is monotone with respect to Q; i.e., if $Q_1 \leqq Q_2$ on partition space $(\mathcal{P}(S), \sqcap)$, then

$$\left\{ f^{-1}[r] : f \in \mathcal{F}, r \in Q_1 \right\} \sqsubseteq \left\{ f^{-1}[r] : f \in \mathcal{F}, r \in Q_2 \right\}$$

on the partially ordered set $(pow(S), \sqsubseteq)$. Since *split*% is monotone in both arguments (cf. Proposition 2.7), and since monotonicity is closed under composition, we know that expression $Q\ split\% \left\{ f^{-1}[r] : f \in \mathcal{F}, r \in Q \right\}$ is monotone in Q. Since *split*% is deflationary in its first argument (cf. Proposition 2.7), then we can use a well-known lattice-theoretic identity (cf. [21] or Theorem 5 in [12]) to rewrite specification (3.2) as,

$$\mathrm{gfp}\, Q . P \sqcap Q\ split\% \left\{ f^{-1}[r] : f \in \mathcal{F}, r \in Q \right\}$$

which can be transformed into the following more convenient form,

$$\mathrm{gfp}\, Q . Q\ split\% \left(P \cup \left\{ f^{-1}[r] : f \in \mathcal{F}, r \in Q \right\} \right) \tag{3.3}$$

using the definition of meet and Proposition 2.7.

Since *split*% and union are monotone in both arguments, and monotonicity is closed under composition, we know that function

$$F(Q) \stackrel{\mathrm{def}}{=} Q\ split\% \left(P \cup \left\{ f^{-1}[r] : f \in \mathcal{F}, r \in Q \right\} \right)$$

is monotone on the partition semilattice $(\mathcal{P}(S), \sqcap, \{S\})$. Hence, specification (3.3) is well defined, and is computed by $F^j(\{S\})$ for some $j \leqq \#S$, according to Theorem 2.2. In [46] Kanellakis and Smolka used a similar approach to obtain an algorithm that ran in time $O(mn)$, where $n = \#S$ and $m = \sum_{f \in \mathcal{F}} \#f$.

3.2. Dominated Convergence

The sequence of iterates (that results from Theorem 2.2) computed by Kanellakis and Smolka's algorithm converges rapidly, but each successive iterate involves potentially costly redundant computations. The main point of dominated convergence is to avoid this redundancy by converging less rapidly towards the fixed point, but proceeding with much more efficient small steps. Instead of computing the expensive and potentially redundant $F(Q)$ operation each step of the Tarski sequence, dominated convergence will form a sequence of small steps with inexpensive *split* operations. Such an approach leads to the efficient algorithm found in [60]. Immediate application of Corollary 2.8 allows us to compute specification (3.3) with the code shown in Figure 3.

```
-- maintain invariant F(Q) ≦ Q
Q := {S}
while ∃ q ∈ (F(Q) − Q) loop
    -- execution frequency O(n)
    Q := Q split q
end loop
```

Figure 3. First approximation to a relational coarsest partition algorithm.

The code found in Figure 3 computes the fixed point through a sequence of inexpensive *split* operations, but the costly calculation of $F(Q)$ is merely shifted to the top of the while-loop. We will avoid this calculation by using the finite differencing transformation to maintain and exploit the invariant

$$T = F(Q) \tag{3.4}$$

However, in order to apply finite differencing conveniently, a small detail needs to be added to our code. Since the code found in Figure 3 maintains the invariant $F(Q) \leqq Q$, we know that each time block q is chosen from $F(Q)$ at the top of the while-loop, there must be a unique block $b \in Q$ such that $q \subset b \wedge q \neq b$. We need to compute b explicitly in the program. Since the choice of q is arbitrary, we can first choose any $b \in Q - F(Q)$, and then base our choice of q on any block within $F(Q)$ that is also contained in b. The result is shown in Figure 4.

```
-- maintain invariant F(Q) ≦ Q
Q := {S}
while ∃ b ∈ (Q − F(Q)), ∃ q ∈ F(Q)|q ⊂ b loop
    -- execution frequency O(n)
    Q := Q split q
end loop
```

Figure 4. Algorithm just before finite differencing.

Invariant (3.4) can be established by executing the assignment,

$$T := F(\{S\}) \tag{3.5}$$

on entry to the loop. After being spoiled by the assignment,

$$Q := Q \; split \; q$$

the invariant can be reestablished by updating T in the following way,

$$T := T\,split\%\,(\{f^{-1}[q] : f \in \mathcal{F}\} \cup \{f^{-1}[b-q] : f \in \mathcal{F}\}) \tag{3.6}$$

just after Q is updated. Code (3.6), which updates T, is derived from the Finite Difference Schema of Lemma 2.10 using the differential identity

$$F(Q\,split\,q) = F(Q)\,split\%\,(\{f^{-1}[q] : f \in \mathcal{F}\} \cup \{f^{-1}[b-q] : f \in \mathcal{F}\}) \tag{3.7}$$

Consequently, we can replace both occurrences of $F(Q)$ within the while-loop by occurrences of T.

It was a variant of differential identity (3.7) that caused trouble for students in the honors section of a graduate course in compilers, which is what motivated the current work. We accept the identity at this point. In the next section the identity is proved formally, and its passage through the NAP proof checker is described.

The decision to eliminate the repeated computation of $F(Q)$ has now been made. The cost of the whole algorithm is now dominated by the cumulative cost of splitting partition T. Recall from the preceding section that with an appropriate implementation the cost of computing $T\,split\,f^{-1}[q]$ is dominated by the cost of computing $f^{-1}[q]$, which is

$$O(\#q + \#\{[x,y] \in f | y \in q\}) \tag{3.8}$$

It is reasonable to reduce this cost by choosing each block q (used in splitting) wisely. One heuristic is to try to lower the sum of the cardinalities $\#q$ over every block q that is chosen by the algorithm. A way to do this was discovered by Hopcroft (see [44]) in the design of his algorithm to compute the minimum state DFA, and adapted by Paige and Tarjan to their relational coarsest partition algorithm; see [60].

Known as Hopcroft's "process the smaller half strategy," the idea is deceptively simple. From the definition of partition refinement, we know that whenever block q is chosen from "thin" partition $T = F(Q)$, the block $b \in Q$ (which contains q) from the "fat" partition must also contain another "thin" block q'. Paige and Tarjan's form of Hopcroft's idea was to examine two arbitrary "thin" blocks belonging to b, and to choose one that contains no more than half the number of elements in b. These modifications are incorporated into the program found in Figure 5.

The significance of Hopcroft's strategy is clear. The sequence $q_1, q_2, \ldots$ of "thin" splitter blocks chosen by the program in Figure 5 satisfies the condition that for every $i = 1, \ldots$ and every $j > i$, either $q_i \cap q_j = \{\}$ or $q_j \subset q_i \wedge \#q_j \leqq \#q_i/2$. Hence, partial order $(\{q_i : i = 1, \ldots\}, \subseteq)$ has the structure of a forest in which the set of all blocks at equal depth in the forest partition a subset of S. Let Q_d be the set of splitters occurring in the forest at depth d. Let $n = \#S$, let $k = \#\mathcal{F}$, and let m be the sum of the total number of pairs in each relation f over all relations contained in $\mathcal{F}$. Then the total cost of splitting by operations of the form $T\,split\,f^{-1}[q]$ for all splitters $q \in Q_d$ at depth d is,

```
-- maintain invariant T ⊆ Q
Q := {S}
T := F({S})
while ∃ b ∈ (Q − T), ∃ q ∈ T | q ⊂ b ∧ #q ≦ #b/2 loop
    Q := Q split q
    T := T split% ({f⁻¹[q] : f ∈ F} ∪ {f⁻¹[b − q] : f ∈ F})
end loop
```

Figure 5. Algorithm after finite differencing.

$$
\begin{aligned}
& O\left(\sum_{f\in\mathcal{F}}\sum_{q\in Q_a} \#q + \#\{[x,y]\in f \mid y\in q\}\right) \\
&= O\left(\sum_{f\in\mathcal{F}}\left(\sum_{q\in Q_a}\#q\right) + \#\{[x,y]\in f \mid y\in \cup_{q\in Q_a} q\}\right) \\
&= O\left(\sum_{f\in\mathcal{F}} \#S + \#\{[x,y]\in f \mid y\in S\}\right) \\
&= O(kn+m)
\end{aligned}
$$

Also, the forest depth cannot exceed $\log\ n$, since the geometric progression $n/2^i$ for $i = 1\ldots$ collapses to 1 in $\log\ n$ steps. Therefore, the cumulative cost of splitting T by operations of the form $T\ split\ f^{-1}[q]$ is $O((kn+m)\log\ n)$. Unfortunately, this strategy alone gives no such bound on operations of the form $T\ split\ f^{-1}[b-q]$, which also appears in the code that updates T. Nevertheless, another strategy found in [60] is used to compute both splitting operations in time (3.8).

Several observations can be made about the program in Figure 5. The code (3.5) used to establish invariant T can be improved by straightforward simplification. Using Proposition 2.7 (4) and (5), we obtain

$$
\begin{aligned}
F(\{S\}) &= \{S\}\ split\%\ (P \cup \{f^{-1}[r] : f\in\mathcal{F}, r\in\{S\}\}) \\
&= \{S\}\ split\%\ (P \cup \{f^{-1}[S] : f\in\mathcal{F}\}) \\
&= (\{S\}\ split\%\ P)\ split\%\ \{f^{-1}[S] : f\in\mathcal{F}\} \\
&= P\ split\%\ \{f^{-1}[S] : f\in\mathcal{F}\}
\end{aligned}
$$

which is computable in $O(kn+m)$ time by previous analysis. Later, we will show how to reduce this cost to $O(n+m)$.

The error published in [60] occurred in the preprocessing steps to establish the initial double partition T and Q. Finite differencing forces the preprocessing and maintenance of T to serve the unified goal of establishing and maintaining a single invariant, which makes errors of that kind less likely. It is fair to say, though, that such errors are regarded as minor in the algorithms community, and are routinely corrected. The benefits of our transformational approach may pertain more to program development than to algorithm design. Yet in the next subsection, we see how straightforward application of our transformations leads to an improvement to Hopcroft's DFA minimization algorithm.

3.3. DFA Minimization

In [2] the problem of minimizing the number of states in a deterministic finite state automaton is formulated exactly like specification (3.1), except that the family $\mathcal{F}$ of relations is restricted to single-valued functions each totally defined on S. Since the many function coarsest partition problem is just a special case of the many relation coarsest partition problem, the program shown in Figure 5 will solve both problems.

The restriction to total functions leads to a startling simplification in the preprocessing operations that establish invariant (3.4). Because each function $f \in \mathcal{F}$ is totally defined on S, we know that $f^{-1}[S] = S$; i.e., the preimage of a function's range is its whole domain. Consequently, the right-hand side of the preprocessing code,

$$T := P \mathit{split}\% \left\{ f^{-1}[S] : f \in \mathcal{F} \right\} \tag{3.9}$$

can be simplified, first to $P \mathit{split}\% \{S\}$, and then directly to P by Proposition 2.7. What this means is that all we have to do is read the input, store it in variable T (i.e., make T an alias of P), and the invariant $T = F(\{S\})$ is established for free.

The restriction to total functions is not exploited in the remainder of the derivation. If partial functions would be allowed, then the preprocessing code (3.9), which takes $O(kn + m)$ to compute, will establish the initial double partition for partial functions, as it did for relations. The remainder of the derivation exploits single-valuedness of relations, and is correct for partial as well as total functions.

We have already discussed one aspect of Hopcroft's "process the smaller half" strategy. This strategy has a second aspect that works for functions (but not relations), and that was exploited in Hopcroft's DFA minimization algorithm; see [44]. Hopcroft's idea is based on the simplest form of Proposition 2.7 (9). That is, if R is a partition of S, $A \subseteq S$ such that $R \mathit{split}\, A = R$, and $B \subseteq A$ such that $B \neq A$, then

$$R \mathit{split}\% \{B, A - B\} = R \mathit{split}\% B = R \mathit{split}\% (A - B)$$

By the way that $F(Q)$ is defined, we know that for any $b \in Q$ and any $f \in \mathcal{F}$, we have $F(Q) \mathit{split}\% f^{-1}[b] = F(Q)$. Since preimages must be disjoint for any

two distinct elements in the range of a function (which is not true for binary relations), we also have,

$$f^{-1}[b-q] \cap f^{-1}[q] = \{\}$$

for any two sets q and b. Hence, the differential identity (3.7) can be simplified to

$$F(Q\ split\ q) = F(Q)\ split\%\ \{f^{-1}[q] : f \in \mathcal{F}\}$$

so that the update code (3.6) can be simplified to

$$T = T\ split\%\ \{f^{-1}[q] : f \in \mathcal{F}\}$$

With this simplification, the earlier analysis of the cumulative time used in splitting operations on T is $O((kn+m)log\ n)$.

The remainder of the derivation has to do with data structuring. The program shown in Figure 6 consolidates the preceding simplifications, and also implements the *split*% operations in terms of the more primitive *split* operations with which it is defined. This is achieved with a SETL for-loop implementation by application of what E. Schonberg (cited as Deak, [25]) calls a compilation transformation.

The space improvement to Hopcroft's algorithm is achieved as follows. Represent the "thin" partition T as the preimages of a single-valued map $T1$ (as described in the last section); i.e., so that identity

$$T = \{T1^{-1}\{q\} : q \in \mathsf{range}\ T1\}$$

holds. Represent "fat" partition Q indirectly by the preimages of a single-valued map $Q1$ defined on the range of $T1$; i.e., so that identity

$$Q = \{T1^{-1}[Q1^{-1}\{b\}] : b \in \mathsf{range}\ Q1\}$$

holds. If we also replace $Q - T$, which appears at the top of the while-loop, by the set $\{b \in \mathsf{range}\ Q1 | \#Q1^{-1}\{b\} > 1\}$ of "fat" blocks that contain more than one "thin" block, we obtain the essence of the double partition implementation used in [60]. The code that results from this shift in representation is shown in Figure 7 and Figure 8.

The final algorithm is obtained by inputting the set $\mathcal{F}'$ of inverse maps (instead of $\mathcal{F}$), and by maintaining and exploiting two more invariants throughout the code in Figure 7 and Figure 8. The first is a map

$$count(q) = \#T1^{-1}\{q\}$$

of the sizes of each "thin" block q; the second is the set

$$ready = \{b \in \mathsf{range}\ Q1 | \#Q1^{-1}\{b\} > 1\}$$

The type assignments for the two new variables are

```
-- maintain invariant T ≦ Q
Q := {S}
-- input partition, and establish invariant T = F({S})
while ∃ b ∈ (Q − T), ∃ q ∈ T|q ⊂ b ∧ #q ≦ #b/2 loop
    Q := Q split q
    for f ∈ F loop
        T := T split f⁻¹[q]
    end loop
end loop
```

Figure 6. The result of "processing the smaller half."

$count : smap(thin - strong, INT)$
$ready : set(fat - strong)$

Just before a value x is removed from a *thin* block by the modification $T1(x)$ $:= om$, *count* is decremented by executing $count(T1(x)) \; -:= 1$. Just before a *thin* block is augmented by the assignment $T1(x) := q$, count is incremented by executing $count(q) \; +:= 1$. Each time a call $q := new(thin)$ creates a new *thin* block q, *count* is initialized by executing $count(q) := 0$. Variable *ready* is initialized to the empty set. After that, it must be maintained relative to two operations. Just before a thin block is removed from a fat block by the assignment $Q1(q) := om$, we perform the conditional update,

```
if #Q1⁻¹{Q1(q)} = 2 then
    ready less := Q1(q)
end if
```

Just before a thin block is added to a fat block by the assignment $Q1(q) := b$, we perform the conditional update,

```
if #Q1⁻¹{b} = 1 then
    ready with := b
end if
```

The space improvement arises from keeping the size of the two auxiliary sets *ready* and *splitter* low, essentially $O(n)$. Hopcroft's workset was of size $\theta(m)$ in the worst case.

Although the algorithm just developed utilizes space efficiently for certain auxiliary data structures, it also suffers from two sources of inefficiency. First, the array representation of $f^{-1} \in \mathcal{F}'$ may have empty components. When $\#\mathcal{F}'$ is large and partial functions are allowed, then the space needed to store all these sparse maps may be excessive.

The second inefficiency is in the repeated computation of argument $f^{-1}[q]$, which is passed to the *split* routine at the bottom of Figure 7. This computation involves iteration through q even though $\mathsf{range}\, f$ may intersect with only a small subset of q. This problem contributes $O(kn\ log\ n)$ steps to the running time.

We end this section with a solution to both problems, and an illustration of how well our type system adds computational transparency to a set-theoretic language. Instead of using a collection $\mathcal{F}'$ of arrays to store the inverses of functions that belong to $\mathcal{F}$, we will use a partial function

$$\mathcal{G} : smap(node - strong, mmap(symb, node))$$

defined by the rule,

$$\mathcal{G}(x) = \{[f, y] : f \in \mathcal{F}, y \in f^{-1}\{x\}\}$$

for every $x \in S$. Base type *symb* represents an alphabet with one symbol for each distinct function that belongs to $\mathcal{F}$. Only list structures are used to implement $\mathcal{G}$, and our algorithm will make no use of arrays. The space needed to store $\mathcal{G}$ and base *symb* is $O(m)$ and $O(k)$ respectively.

Based on the definition of $\mathcal{G}$, we know that the following two identities,

$$\begin{aligned} f^{-1}\{x\} &= \mathcal{G}(x)\{f\} \\ f^{-1}[q] &= \{y : x \in q, y \in \mathcal{G}(x)\{f\}\} \end{aligned}$$

hold for any "thin" block q and any function $f \in \mathcal{F}$. Hence, for fixed q we can compute auxiliary map $bucket : mmap(symb - strong, set(node))$ by the assignment,

$$bucket := \{[f, \mathcal{G}(x)\{f\}] : x \in q, f \in \mathsf{domain}\, \mathcal{G}(x)\}$$

in order to encode preimage sets $f^{-1}[q]$ for all $f \in \mathcal{F}$; i.e.,

1. $\mathsf{domain}\, bucket = \{f \in \mathcal{F} | \mathsf{range}\, f \cap q \neq \{\}\}$,
2. $bucket\{f\} = \{f^{-1}\{x\} : x \in q\}$ for every $f \in \mathsf{domain}\, bucket$, and, hence,
3. $f^{-1}[q] = \{y : T \in bucket\{f\}, y \in T\}$ for every $f \in \mathcal{F}$.

The preceding analysis allows us to replace the iterated calls to *split* at the bottom of Figure 7 by the equivalent, but more efficient, code shown in Figure 9.

The cost of computing *bucket* is $O(\#q + \sum_{x \in q} \#\{f \in \mathcal{F} | x \in \mathsf{range}\, f\}$. The cost of the for-loop through $\mathsf{domain}\, bucket$ is dominated by the cumulative cost of computing $f^{-1}[q]$ for every $f \in \mathsf{domain}\, bucket$. This cost is $\sum_{f \in \mathcal{F}} \#f^{-1}[q]$. Consequently, the total running time of the algorithm is $O((m+n)log\ n)$. The space needed to store the new auxiliary data *bucket* is $O(\sum_{x \in q} \#\{f \in \mathcal{F} | x \in \mathsf{range}\, f\})$, which starts out being as much as $O(m/2)$ in the worst case, and gradually diminishes as "thin" blocks q get progressively smaller.

As a final clarification to the algorithmic analysis, we note that the preprocessing operations will partition the set $\{x \in S | (\forall f \in \mathcal{F} | x \notin \mathsf{domain}\, f)\}$ into

```
P : set(set(node)) -- the read method ensures that P is a partition
T1 : smap(node − strong, thin − strong)
Q1 : smap(thin − strong, fat − strong)
F' : set(mmap(node − strong, node)) -- set of inverse maps
-- input conversion
b := new(fat)
for p ∈ P loop
    q := new(thin)
    Q1(q) := b -- corresponds to Q := {S}
    for x ∈ p loop
        -- execution frequency n
        T1(x) := q -- corresponds to T := P
    end loop
end loop
-- repeated partition refinement
while ∃b ∈ {b ∈ range Q1 | #Q1^-1{b} > 1} loop
    q1 := ∋ Q1^-1{b}
    q2 := ∋ (Q1^-1{b} less q1)
    if #q1 ≦ #q2 then
        q := q1
    else
        q := q2
    end if
    Q1(q) := om -- remove q from b
    Q1(q) := new(fat) -- make q a new 'fat' block
    for f^-1 ∈ F' loop
        split(T1, f^-1[q])
    end loop
end loop
```

Figure 7. Double partition algorithm.

```
-- call by reference
procedure split(rw T1 : smap(node − strong, thin − strong), splitter : set(node))
twin : smap(thin − strong, thin)
twin := {}
for x ∈ splitter loop
    -- execution frequency #splitter
    qold := T1(x)
    qnew := twin(qold)
    if qnew = om then
-- generate a new base element that 'names' the new block
        qnew := new(thin)
        Q1(qnew) := Q1(qold)
        twin(qold) := qnew
    end if
    if # T1^{-1}{T1(x)} = 1 then
        Q1(T1(x)) := om
    end if
-- indexed assignment to T1 is broken into two operations
    T1(x) := om      --remove x from block T1^{-1}{qold}
-- we could execute delete(qold) if T1^{-1}{qold} = {}
    T1(x) := qnew    --add x to block T1^{-1}{qnew}
end loop
end split
```

Figure 8. Double partition algorithm.

separate blocks P' that will never be split again. The worst case time to do this is $O(m+n)$. Subsequently, each block in P' will be used at most once for splitting other blocks. The total cost of such splitting is also $O(m+n)$. As was observed in [60], the number n' of elements in S outside of P' is at most m. Hence, if we prune blocks that belong to P' from the forest (used in earlier analysis of the "process the smaller half" strategy), then a more accurate time bound for our algorithm is $O(n + m + m\ log\ n)$. The read method for constructing the initial data structures is left as an exercise.

Cardon and Crochemore used another strategy based on lexicographical sorting and on exploiting Proposition 2.7 (9) to obtain a DFA minimization algorithm with similar asymptotic time and space bounds; see [14]. However, a finer parameterization would reveal that our algorithm has a theoretical and computational advantage in space and time.

```
-- low level code to compute bucket
bucket := {}
for x ∈ q loop
    -- execution frequency #q
    for f ∈ domain G(x) loop
        -- execution frequency Σ_{f∈F} #(q ∩ range f)
        bucket{f} with:= G(x){f}
    end loop
end loop
-- iterated call to split
for f ∈ domain bucket loop
    -- execution frequency #{f ∈ F | range f ∩ q ≠ {}}
    splitter := {}
    for T ∈ bucket{f} loop
        -- execution frequency Σ_{f∈F} #(q ∩ range f)
        for x ∈ T loop
            --execution frequency Σ_{f∈F} #f^{-1}[q]
            if x ∉ splitter then
                splitter with:= x
            end if
        end loop
    end loop
    split(T1, splitter)
    bucket(f) := om -- initialize bucket
end loop
```

Figure 9. Splitting with sparse data structures.

4. Mechanical Verification of a Finite Difference Transformation

In this section we go through the main steps of how the differential identity (3.7) that supported the finite difference transformation in the last section was formally proved within the NAP proof checking system. This requires a brief discussion of the lattice properties of the set $\mathcal{P}(S)$ of all partitions over S. It includes a systematic exposition of definitions, corollaries and simple lemmas, which reproduces exactly the structure of the proof accepted by the NAP system.

Unlike a mathematician, who is likely to be satisfied with a less formal proof or even a proof sketch, the NAP proof checker does not allow unstated hypotheses, or proof gaps. All the elementary theorems in number theory, and naive set theory needed in the proofs have to be formally expressed if not proved. We will state precisely all the definitions we had to introduce, and all the derived

corollaries and lemmas. The background theory out of which arithmetics and naive set theory are obtained is Hereditarily Finite Set Theory, which is assumed to be the semantic foundation of SETL and its Cantor dialect. The NAP proof checker uses the Cantor runtime system to directly evaluate the validity of simple Boolean identities involving either constants or terms built out of constants (e.g., $\#\{1,3\} = 2$).

It is useful to restate some of the basic definitions and elementary properties that were introduced earlier.

DEFINITION 4.1. *If S is a finite set, then P is a partition of S if and only if*

1. $P \subseteq pow(S) - \{\{\}\}$ *(P is a collection of nonempty subsets of S.)*
2. $\cup/P = S$ *(S is the union of all the sets in P.)*
3. $\forall p, q \in P | p \neq q \rightarrow p \cap q = \{\}$ *(The elements of P are pair-wise disjoint.)*

DEFINITION 4.2. *The set $\mathcal{P}(S)$ of all partitions over S may be partially ordered as follows: If $Q, P \in \mathcal{P}(S)$, then we say that Q is a refinement of P, denoted by $Q \leqq P$, if and only if $\forall q \in Q | (\exists p \in P | q \subseteq p)$. The partially ordered set $(\mathcal{P}(S), \leqq)$ has maximum element $\{S\}$ and minimum element $\{\{x\} : x \in S\}$.*

DEFINITION 4.3. *If P is a partition of a finite set S, and $a \subseteq S$ (i.e., $a \in pow(S)$). Then*

$$P \text{ split } a \stackrel{\text{def}}{=} \{s \in pow(S) | s \neq \{\} \wedge (\exists p \in P | s \in \{p \cap a, p - a\})\}$$

PROPOSITION 4.4. *Let P and Q be two partitions of finite set S, and let $a, b \subseteq S$.*

1. *P split $a \leqq P$ (split is deflationary in its first argument)*
2. *If $P \leqq Q$, then P split $a \leqq Q$ split a (split is monotone in its first argument)*
3. *P split $a = P$ if and only if $\exists A \subseteq P | a = \cup/A$ (splitting a partition has no effect if and only if one splits by any union of elements of P)*
4. *P split $a = P$ if and only if $\forall p \in P | p \subseteq a \vee p \cap a = \{\}$*
5. *$a = \cup/\{x \in P \text{ split } a | x \subseteq a\}$ (Set a is exactly covered by a subset of P split a)*
6. *If $p \in P$, $a \subseteq p$, $a \neq p$, and $a \neq \{\}$, then P split $a = (P - p) \cup \{a, p - a\}$ (Splitting P by a set contained within some block $p \in P$ is a local change in which p is replaced by a and $p - a$.)*
7. *If $A \subseteq P$, $A \neq \{\}$, $a = \cup/A$, $b \subset a$, and $b \neq a$, then P split $b = (P - A) \cup A$ split b (Splitting by a small set affects only the blocks that intersect that set.)*

8. $(P \; split \; a) \; split \; b = (P \; split \; b) \; split \; a$ *(The result of splitting with respect to more than one set is the same regardless of the splitting order.)*

A useful, simple consequence of Proposition 4.4 (6) is: if P is a partition of S, $a \in P$, $b \subset a$, and $b \notin P$, then

$$P \; split \; b = P \; split \; (a - b)$$

Throughout this section every function $f : S \rightarrow S$ and relation $E \subseteq S \times S$ has S as its range and domain.

The first major step in the proof of differential identity (3.7) is to provide a detailed account of the relationship between the split operation and the meet operation over the $\mathcal{P}(S)$ semilattice.

DEFINITION 4.5. (COMPOUND OPERATIONS) *Let A, B be sets, and let F be a binary operation $A \times B \rightarrow A$. Let $a \in A$, and let t be an ordered subset of B. Then $a \; F\% \; t$ is defined recursively as:*

1. $a \; F\% \; [] = a$
2. $a \; F\% \; u \; with \; b = (a \; F\% \; u) \; F \; b$

We will formally define compound operations over unordered collections in the following way. If for every two arbitrary enumerations t, t' of the same subset of B, we can prove

$$a \; F\% \; t \; = \; a \; F\% \; t'$$

then we define the following equivalence relation over ordered subsets of B:

$$t \; \equiv \; t' \text{ if and only if } \{x : \; x \; \in \; t\} = \{x : \; x \; \in t' \; \}$$

The equivalence classes of this relation correspond to subsets of B. In this way we extend the definition of $F\%$ to unordered sets, and formally justify Definition 2.6 of the compound operation *split*% applied to sets. The following corollary characterizes Definition 4.5 of the compound operation formed from the *split* operation introduced in Definition 4.3.

COROLLARY 4.6. (THE COMPOUND SPLIT OPERATION ON ORDERED SETS) *Let P be a partition of S, and let t be an ordered collection of subsets of S. Then $s \in P \; split\% \; t$ if and only if*

1. $s \; \neq \; \{\}$
2. *there is a tuple v of subsets of S such that:*
   $\#v \; = \; \#t + 1, \; v(1) \; \in \; P, \; v(\#v) \; = \; s,$ *and*
   $\forall \; i \; \in \; [1...\#t] \mid v(i+1) \; \neq \; \{\} \; \wedge \; v(i+1) \; \in \; \{v(i) \cap t(i), \; v(i) - t(i)\}$

The proof of Corollary 4.6 is a straightforward induction over the length of t. It is interesting to note that the formal statement of Corollary 4.6 is the result of many "approximations," i.e., iterations over versions differing very slightly from one another in their wording. It was only while pursuing the systematic proof construction with NAP's help that the defects of the successive versions were revealed. The induction proof contains around four hundred elementary proof steps.

Corollary 4.6 has several consequences, which are mentioned below.

COROLLARY 4.7. (CHAINING) *Let P, S, t be as in Corollary 4.6. If v is the tuple associated with s in P split% t, then*

1. $\forall\ i\ \in\ [1..\#t] \mid v(i+1)\ \subset\ v(i)$
2. $\forall\ i\ \in\ [1..\#t] \mid v(i+1)\ \subset\ v(1)$
3. *if* $\exists\ j\ \in\ [1..\#t] \mid v(j+1)\ =\ v(j) \cap t(j)$ *then*
   - $\forall\ i\ \in\ [1..\#t+1] \mid i > j\ \rightarrow\ v(i) \subset t(j)$
     (N.B. $i\ \in\ [1..\#t+1]$ *if and only if* $i \in\ [1..\#v]$*)*
   - $s\ \subset\ t(j)$

COROLLARY 4.8. (COMPOUNDING SPLIT WITH A PARTITION — 1) *Let P, S, t be as in Corollary 4.6. If v be the tuple associated with s in P split% t, then*

1. *If the elements of t are mutually disjoint, then*
   there is at most one j: $j\ \in\ [1..\#t] \mid v(j+1)\ =\ v(j) \cap t(j)$
2. *If* $\cup/t\ =\ \cup/P\ =\ S$, *then*
   there is at least one j: $j\ \in\ [1..\#t] \mid v(j+1)\ =\ v(j) \cap t(j)$

(*N.B.* (1) *and* (2) *are independent conditions entailing independent conclusions.*)

If t satisfies the conditions (1) and (2) of the previous corollary, then $t's$ members form a partition of S. We will then loosely state that *t is a partition of S.*

COROLLARY 4.9. (COMPOUNDING SPLIT WITH A PARTITION — 2) *Let P, S, t be as in Corollary 4.6, and let t be a partition of S. Then the converse of the previous corollary holds too; i.e.,*

$$s \in P \text{ split\% } t \text{ if and only if } \exists\ p\ \in\ P, q\ \in\ t \mid s\ =\ p \cap q$$

COROLLARY 4.10. (COMPOUNDING SPLIT WITH A PARTITION — 3) *Let P, S be as in Corollary 4.6. If t and t' are two enumerations of the same partition of S, then*

$$s\ \in\ P \text{ split\% } t \text{ if and only if } s\ \in\ P \text{ split\% } t'$$

The preceding corollary justifies an equivalence relation over ordered enumerations of the elements of partitions:

$$t \equiv t' \text{ if and only if } t \textit{ and } t' \textit{ enumerate the same partition } R \textit{ of } S$$

Consequently *split*% is well-defined for partitions of S:

$$P \textit{ split\% } R = P \textit{ split\% } t$$

where t enumerates R.

DEFINITION 4.11. (MEET OPERATION) *Operation meet over partitions of S is defined as*

$$P \textit{ meet } R \stackrel{\text{def}}{=} P \textit{ split\% } R$$

This definition is motivated by the following corollary:

COROLLARY 4.12. *Operation meet satisfies:*

- $Q \leqq A \wedge Q \leqq B$ *if and only if* $Q \leqq A$ *meet* B (meet takes its name from this property)
- P *meet* $P = P$ (idempotence)
- P *meet* $R = R$ *meet* P (commutativity)
- $(P$ *meet* $R)$ *meet* $U = P$ *meet* $(R$ *meet* $U)$ (associativity)

Corollary 4.12 entails that the *meet* of any collection of partitions is well-defined, independently of any ordering of the collection, and has exactly the property of the meet operation $\sqcap$ over a semilattice. Consequently, compound split is well-defined not only for ordered collections of subsets of S but for unordered collections too (see Definition 4.16 below).

DEFINITION 4.13. (BASE PARTITION) *Let $a \subset S$. Consider the mapping $\mathcal{Q}$: $pow(S) \rightarrow \mathcal{P}(S)$ defined by the rule,*

$$\mathcal{Q}(a) \stackrel{\text{def}}{=} \begin{array}{ll} \{a, S-a\} & \texttt{if } a \neq \{\} \texttt{ and } a \neq S \\ \{S\} & \texttt{if } a = \{\} \texttt{ or } a = S \end{array}$$

COROLLARY 4.14. *If P is a partition of S, and a is a subset of S, then*

$$P \textit{ split } a = P \textit{ split\% } \mathcal{Q}(a)$$

COROLLARY 4.15. (COMPOUNDING SPLIT WITH A SET — 1) *Let P be a partition of S, and let X be a set of subsets of S (i.e., $X \subseteq pow(S)$). If t is a tuple enumerating the elements of X, then*

$$P \; split\% \; t \;=\; P \sqcap\% \; [\mathcal{Q}(x) : x \;\in\; t]$$

The proof of Corollary 4.15 is by induction on $\#X$.

Given the associative and commutative properties of $\sqcap$, and since t enumerates X, we have

$$P \sqcap\% \; [\mathcal{Q}(x) : x \in t] \;=\; P \sqcap\% \; \{\mathcal{Q}(x) : x \in X\}$$

If t and t' are two arbitrary enumerations of X, then

$$P \; split\% \; t \;=\; P \sqcap\% \; \{\mathcal{Q}(x) : \; x \;\in X\} = \; P \; split\% t'$$

DEFINITION 4.16. (COMPOUNDING SPLIT WITH A SET — 2)
If P is a partition of S, and X is a set of subsets of S (i.e., $X \subseteq pow(S)$), then

$$P \; split\% \; X = P \sqcap\% \; \{\mathcal{Q}(x) : \; x \;\in X\}$$

A simple application of Definition 4.16 is :

COROLLARY 4.17. (SPLITTING WITH TWO ARBITRARY SETS)
If P is a partition of S, and a,b are any two subsets of S, then

1. $\mathcal{Q}(a \cup b) \sqcap \mathcal{Q}(a) \sqcap \mathcal{Q}(b) \;=\; \mathcal{Q}(a) \sqcap \mathcal{Q}(b)$
2. $P \; split\%\{a \cup b, a, b\} = P \; split\%\{a, b\}$

Note that (1) above may be easily verified directly, e.g., using

$$\mathcal{Q}(a \cup b) \sqcap \mathcal{Q}(a) \sqcap \mathcal{Q}(b) = \mathcal{Q}(a \cup b) \; split\% \; \{a, b\}$$

and

$$\mathcal{Q}(a) \sqcap \mathcal{Q}(b) = \mathcal{Q}(a) \; split \; b$$

However a direct verification of (2) without the use of Definition 4.16 does not seem that obvious!

Another consequence of Definition 4.16 is:

COROLLARY 4.18. *If P is a partition of S, and X,Y are two arbitrary subsets of S, then*

$$P \; split\% \; (X \cup Y) = (P \; split\% \; X) \sqcap (P \; split\% \; Y)$$

Mappings from S into S have simple categorical properties with regard to the meet operation. Let us briefly recall some elementary properties of inverse mappings. Let f: $S \rightarrow S$, and let $a \subseteq S$. Then,

$$f^{-1}[a] \;=\; \{x : \; x \;\in\; S \mid f(x) \;\in\; a\}$$

A useful property of f^{-1} is: Let f: $S \rightarrow S$, and let P be a partition of S. Then

$$\{f^{-1}[q] : \; q \;\in\; P \mid f^{-1}[q] \;\neq\; \{\}\}$$

is a partition of S.

This justifies the following notation.

DEFINITION 4.19. (INVERSE-PARTITIONING) *Let f: $S \rightarrow S$, and let P be a partition of S.*

$$f^{-1}(P) \;\stackrel{\text{def}}{=}\; \{f^{-1}[q] : \; q \in P \mid f^{-1}[q] \;\neq\; \{\}\}$$

This definition has the following consequences.

COROLLARY 4.20. *Let f: $S \rightarrow S$, and let $a \subset S$. Then,*

$$f^{-1}(\mathcal{Q}(a)) \;=\; \mathcal{Q}(f^{-1}[a])$$

COROLLARY 4.21. *Let f: $S \rightarrow S$, and let P and R be partitions of S. Then,*

$$f^{-1}(P \sqcap R) \;=\; f^{-1}(P) \sqcap f^{-1}(R)$$

We will proceed to prove differential identity (3.7) by exploiting properties of *split*% in combination with properties of inverse mappings.

COROLLARY 4.22. *Let g be a mapping pow(S) $\rightarrow$ pow(S). Let P,Q be partitions of S. Let G be defined by the rule*

$$G(Q) \;\stackrel{\text{def}}{=}\; P \; \textit{split}\% \; \{g(r) : r \;\in\; Q\}.$$

Let b in Q - G(Q) , $a \subset b$, $a \notin Q$. If

$$g(b) \;=\; g(a) \;\cup\; g(b - a)$$

then

$$G(Q \; \textit{split} \; a) \;=\; G(Q) \; \textit{split}\% \; \{g(a), g(b - a)\}$$

Examples of such a g are the inverse map f^{-1} for a function f: $S \rightarrow S$, or the inverse relation R^{-1} for a binary relation $R \subseteq S \;\times\; S$ (N.B. $R^{-1}[a] =$

$\{x : [x,y] \in R \mid y \in a\}$). Such functions satisfy $g(p \cup q) = g(p) \cup g(q)$ for arbitrary p, q, not necessarily disjoint.

The differential identity (3.7) follows from the simple topological properties of partitions stated in the lemma just below.

LEMMA 4.23. *If R and Q are two distinct partitions of S, and $R < Q$, then*

1. $q \in Q - R \rightarrow \{x \in R | x \subset q\} \neq \{\}$ *(every element of $Q - R$ contains at least one element of R)*
2. $q \in Q - R \rightarrow \#\{x \in R | x \subset q\} \geqq 2$ *(every element of $Q - R$ contains at least two disjoint elements of R)*
3. $q \in Q - R \rightarrow \exists x \in R | x \subset q \wedge \#x \leqq \#q/2$
4. $\exists r \in R - Q | R \leqq (Q \ split \ r) < Q$
5. $q \in Q - R \rightarrow \exists r \in R | r \subset q \wedge \#r \leqq \#q/2 \wedge R \leqq (Q \ split \ r) < Q$

THEOREM 4.24. (GENERALIZED DIFFERENTIAL IDENTITY)

1. *Let g,P,Q,G be as in Corollary* 4.22. *Let F(Q) = G(Q) $\sqcap$ Q, so that F(Q) $\leqq$ Q. Let F(Q) < Q, and let q and r be sets as those defined by Lemma* 4.23 (4) *or* (5) *applied to Q and R = F(Q), i.e., q ∈ Q - F(Q) and r ∈ F(Q) - Q, r ⊂ q. Then*

$$F(Q \ split \ r) \ = \ F(Q) \ split\% \ \{g(r), g(q - r)\}$$

2. *Let g_i: $i \in [1..k]$ be mappings pow(S) → pow(S). Let P,Q be partitions of S, and let G satisfy*

$$G(Q) \ = P \ split\% \{g_i(r) : r \in Q, \ i \in [1..k]\}$$

Let F(Q) = G(Q) $\sqcap$ Q, so that F(Q) $\leqq$ Q. If F(Q) < Q, let q and r be sets as those defined by Lemma 4.23 (4) *or* (5) *applied to Q and R = F(Q), i.e., q in Q - F(Q) and r ∈ F(Q) - Q, r ⊂ q. If*

$$\forall i \in [1..k] \mid g_i(q) \ = \ g_i(r) \cup g_i(q - r)$$

(as in Corollary 4.22*) then*

$$F(Q \ split \ r) \ = \ F(Q) \ split\% \ (\cup/\{\{g_i(r), g_i(q - r)\} : i \in [1..k]\})$$

5. Overview of NAP

This overview describes the main features of NAP, the system used to construct the formal proof outlined in the last section. For more information see

[48]. NAP is a simple reasoning system for constructing proofs in First Order Logic in either interactive or batch mode. It is based on variants of two earlier systems, FOL, [90], and XCHECK, [83], that evolved during the 1970's and early 1980's. As such, NAP incorporates no novel concept, and it lacks many of the capabilities found in modern reasoning systems such as the Calculus of Constructions, [19], Deva, [78], [5], and [88], ISABELLE, [61], HOL, [38], LP, [37], Nqthm, [8], and many more. NAP was chosen because it was already implemented by the first author, it runs on his Macintosh computer, and it has a syntax compatible with SETL. We were also vaguely skeptical that the capabilities currently incorporated in more sophisticated systems would help in any major way.

We believe that our experiment in using NAP to combine mechanical verification with algorithm design and program development within the derivation of a difficult combinatorial algorithm is novel. There are a few related experiments in the literature in which mechanical verification is combined with transformational programming. One example is Broy's use of Larch (see [41]) and its proof assistant LP (see [37]) to derive a version of Quicksort by transformation; see [9]. Like Broy's example, the other work that we have seen of this kind involves derivations of fairly simple, known algorithms that have been studied extensively.

Because NAP has been designed to prove properties of programs written in Cantor (see [47]), the syntax and semantics of NAP and Cantor are linked. Axioms and theorems in NAP are Cantor Boolean expressions, and are parsed with the Cantor parser. Since NAP is a dialect of SETL, it is specially equipped for reasoning in Hereditarily Finite Set Theory.

Within NAP a formal language is for FOL predicate calculus, and is characterized by its non-logical symbols. Each non-logical symbol has to be declared either as a predicate or a function, and has a specific arity. All undeclared non-logical symbols are assumed to be variables.

Since quantification and set comprehension must be bounded in Cantor (as in the SETL notation described earlier), only safe set expressions are allowed in the sense of [13]. A special constant U that represents an infinite universal set has been introduced exclusively to deal with unbounded iterators or quantifiers. NAP requires $exists\ x\ in\ U \mid K(x)$ and $\{x : x\ in\ U \mid K(x)\}$ instead of $exists\ x \mid K(x)$ and $\{x \mid K(x)\}$.

The well-formed formulas are inductively defined as the smallest set that contains:

- (atomic formulas) $R(t_1, \ldots, t_k)$, where R is a declared predicate of arity k, and $t_1, \ldots, t_k$ are terms,
- Cantor Boolean symbols applied to terms, provided that the axioms defining them are in the theory; e.g.,
  - (in-)equality: =, /=
  - set membership and containment: in, notin, subset, incs (substitutes for $\in$, $\notin$, $\subset$, $\supset$)

  - arithmetic comparison: $<,\ \leqq,\ >, \geqq$

- the meta-formula in one variable, FM(x), which represents a predicate of arity 1, which could be used in axiom schemes, e.g., for stating the Peano induction rule, or the separation axiom of set theory.

Terms are inductively defined as the smallest set that contains:

- $f(t_1, \ldots, t_k)$, where f is a declared function symbol of arity k and $t_1, \ldots, t_k$ are terms,
- variables (declared implicitly),
- constant U (of arity 0), which may appear only as part of an iterator or a Boolean expression $aterm \in U$ (which is always deemed true), and
- Cantor set, tuple, and arithmetic expressions, provided axioms are introduced to define the built-in set and arithmetic functions that appear in those expressions.

For a given formal language, a theory (which should include axioms for all the non-logical operations and formal Boolean tests) is defined by presenting

- a list of abbreviations, i.e., the assignment of names to all the reference formulas, which include the axioms, and theorems that have been derived from these axioms,
- a list of axioms, and
- a list of theorems.

NAP supports natural deduction for proof construction.

- a proof has a goal (the theorem to be proved) stated in a given formal language within a given theory T, i.e., a declared set of axioms and previously derived theorems — presumably a consistent set
- a proof has a derivation, defined as a sequence of derivation steps.

Each proof step may be represented by a *triple*

$$(premises,\ operation,\ resulting\ formula)$$

and each step has a corresponding rank in the derivation sequence (e.g., the first step is always the goal statement). The *current_step_rank* is maintained by the NAP system. In written proof scenarios, *current_step_rank* may be represented by the star symbol "*". The representation by *triples* is a further formalization of the common way to represent a proof as a sequence of formulas:

$$f_1,\ f_2, \ldots, f_n\ =\ theorem$$

where each f_i is either an element of the theory T or is derived by application of a logical rule to formulas in $f_1, f_2, \ldots, f_{i-1}$ or in T. At step i formula f_i results from applying an *operation* of the form

$$op\ arglist$$

where *op* is an operator and *arglist* is a list of arguments. Each argument may refer to formulas in T or to variable names using ranks of resulting formulas from previous steps. For example, there is a rule

$$insert\ fla$$

where *fla* refers to a formula (either an axiom or a previously established theorem) that belongs to T. The formula that results from this *operation* is precisely the text of the given fla.

Another rule has the form

$$modus\_ponens\ rk_a\ rk_{a->b}$$

where rk_a, $rk_{a->b} < current\_step\_rank$ represent formulas of ranks rk_a and $rk_{a->b}$ respectively. If these two formulas conform to the usual modus ponens pattern

$$\begin{array}{l} A \\ A \rightarrow B \end{array}$$

then the resulting formula is B; otherwise, it is undefined.

There are several rules for transforming a formula by substitutions. One such rule is,

$$substitute\_equ\ rk\ x\ =\ aterm$$

where $rk < current\_step\_rank$ designates a formula A obtained at derivation step rk, where x represents a variable occurring free in A, and where *aterm* is substitutable[1] for any free occurrence of x in A. The resulting formula is $A(x/aterm)$, which is obtained by substituting all the free occurrences of x by the term *aterm*. Another substitution rule is

$$replace\_equ\ rk_{eq}\ rk$$

where rk_{eq}, $rk < current\_step\_rank$. Rank rk_{eq} designates an equality or an equivalence formula, and rk is a previously obtained formula which should be transformed according to the substitution defined by equality or equivalence. This substitution requires exact term matching (in the case where rk_{eq} designates an equality) or an exact subformula matching (in the case where rk_{eq} designates an equivalence).

For convenience, working premises may be introduced freely using the operation

$$wp\ aFormula$$

where *aFormula* is either a declared abbreviation or the text of a formula representing a hypothesis. Each such hypothesis will have to be subsequently "cancelled" either by proving it (e.g., as a separate lemma), by deriving a contradiction (i.e., proving its negation), or by deriving an implication. These three cancellation methods appear as proof schemes in Appendix A.

[1] According to [77] "[a term] a is substitutable for x in [formula] A if for each variable y occurring in a, no part of A of the form $\exists\ y\ B[or\ \forall\ y\ B]$ contains an occurrence of x which is free in A."

The set of premises used to derive the resulting formula at each step i is stored in the first component *premises* of the triple representing step i. The set of premises at each step grows by inheritance. For example, if *op* derives formula f from f_a and f_b (of ranks rk_{f_a} and rk_{f_b} respectively) at step rk, then

$$premises(rk) \;=\; premises(rk_{f_a}) \;\cup\; premises(rk_{f_b})$$

where *premises(i)* denotes the premises component of the i-th derivation step.

When at some derivation step of rank $rk > 1$ the resulting formula is the same as the goal (the first formula), and if *premises(rk)* is empty, then the NAP system recognizes that the goal is proved. Otherwise, NAP signals a partial proof completion, which requires that all the statements in *premises(rk)* must be proved.

The "reasoning" power of NAP derives from its set of operations, which may be subdivided into the following categories:

1. the inference rules,
2. the tautological rewriting rules,
3. the predicate calculus substitution rules,
4. specialized set-theoretic rules,
5. the proof schemes.

NAP is essentially a syntactic engine that compares, transforms, and writes well-formed formulas. The rules are therefore transformation rules. For the sake of clarity, they are presented in Appendix A as sequent pattern formation rules.

In all these rules but the proof schemes, the set of premises, which results from applying a rule at a derivation step, grows by inheritance, i.e., is just the union of all the premises of the contributing steps. Hence, the rule that defines the resulting set of premises is omitted. For proof schemes the full premises are given in detail.

The rules found in Appendix A constitute a complete set of rules for predicate calculus. These rules consist of a smaller subset that is complete (see [4]) plus rules that are added for convenience.

The following mechanisms can upset this very standard architecture.

1. A declared theory is not proved sound. It is beyond NAP's power to verify the consistency of submitted theories. NAP only supervises the proof checking operations.
2. A working premise or *wp* operation lets us introduce "statements," i.e., assertions that must be verified by direct evaluation in Cantor. Examples of such statements are

```
wp 1 in {1,2,3};
wp 4 = 2+2;
wp #{} = 0;
wp ['a','b','c'](1) = 'a';
wp forall x in [[],[],[]] | #x = 0;
```

An equivalent feature is described in [24] as part of what is called "the verifier programming environment."

3. It is necessary to introduce new proof rules (new sequent patterns). The rules for predicate calculus are clearly sufficient. However, the specific rules for set theory and for dealing with SETL-like program expressions may prove insufficient or inadequate. Indeed those rules are intended to capture our understanding of set-oriented programming. The introduction of new rules is an easy task from a software viewpoint. However, no generative mechanisms have been introduced into NAP to discourage the introduction of invalid rules. The required verification must be done outside NAP.

There are no other problematic forms of extensibility in NAP. New axioms and new symbols are processed in standard ways — by creating a new theory declared as an extension of a given source theory. The new theory inherits all the theorems of its source. However, conservative extensions are not considered truly new theories, and adding (Henkin) constants does not require creation of an extension. When new function or predicate symbols are added to the formal language, however, an extension needs to be created. And this is just the right thing to do, since adding such meaningful symbols always comes with the definition of new axioms for these symbols.

NAP supports an interactive proof construction process, and a non-interactive one, which we prefer to call a proof programming process. The rules listed in Appendix A are the essential rules in the interactive mode. In that mode NAP favors a transformational approach to pre-existing formulas, and discourages the interactive definition of new formulas, unless it is absolutely necessary. This is because purely syntactic errors, though easily recovered, are sometimes difficult for an interactive user to distinguish from deeper logical errors. Editing formulas by using the formal *substitute_equ* or *replace_equ* operations may be somewhat clumsy, but is entirely safe. Consequently, interactive sessions characteristically have a large number of purely clerical substitution steps.

NAP, drawing heavily on unification, supports forms of Obvious Logical Inferences (see [23]) for reducing the number of substitution steps required in a derivation. This is obtained by specifying for a given operation, not only its "source" formulas and parameters, but its target formula too. Unification of the target formula with the expected result formula for a given rule defines transformations for the source formulas. If these transformations are valid substitutions, NAP performs all the necessary substitutions. Schemes for these generalizations are available for a number of the rules mentioned in Appendix A. These schemes relax all the requirements for exact term or sub-formula matching in the corresponding rules. These facilities are only available in the proof programming process (i.e., in the non-interactive mode).

5.1. Interactive Proof Construction Capabilities

The main proof construction capabilities of NAP are to:

- create a proof interactively, step by step:

  - define a goal (a formula to prove within a given theory)
  - insert axioms , established theorems, or conjectures
  - derive new intermediate proof steps via built-in syllogisms (cf. the tables in Appendix A)
  - introduce formulas, abbreviations, auxiliary symbols, or comments
  - detect full and conditional proof completion
- save (on file) a completed or uncompleted proof session
- review a session: restore (from file) an existing proof, and
  - check it as a whole or by breakpointing one step at a time
  - continue to construct the proof if it is not complete
- produce a session script: the session script contains both the proof script and the proof trace; i.e., the actual list of formulas generated by the succession of derivation operations

Appendix B contains a proof script that illustrates this interactive step-by-step process. The theorem proved in Appendix B is well known, and was proved mechanically before within the XCHECK system; see [83].

5.2. Proof Programming Capabilities

The main programming tools and capabilities of NAP are to:

- specify a theory declaration:
  the declaration consists chiefly in a language definition, i.e., a list of non-logical symbols, a list of abbreviations (i.e., identifiers for specific formulas), and lists of axioms and already established theorems. Axiom and theorem lists are lists of abbreviations terminated by a semi-colon.
- specify a proof script
  - the required syntax is:

```
<proof step> ::= <derivation_operation> <Arglist> ;
                                    [ @ comment_text @ ]
```

  - several quoting forms are needed to distinguish strings from logic assertions, comments, and step labels, e.g.:
    * logic formula

```
$x/=y and Is_a_Partition(p) impl x*y = {} $
```

    * comment

```
@ members of a partition are mutually disjoint @
```

    * label

```
:disjoint_0:
```

  - arguments in *arglist* are formulas, labels, relative addresses (e.g., *-2), integer constants (e.g., the rank of a term)

- compile theory and script into a session: rudimentary syntactic error detection seems sufficient
- review the session: as in the interactive mode
- debug the proof: modify a script, recompile, review

Appendix C contains the theory declaration and proof text used to generate a script equivalent to that shown in Appendix B.

Improving Readability with Obvious Logical Inferences

The available obvious logical inference mechanisms could have shortened the proof in Appendix C. For example, the two instructions at the end of the "esias" lemma:

```
comm_iff_  1 2 :SET_empty: ;
replace_equ * - 1 * - 2;
        @ QED (CQFD)@
```

could be replaced by a single one:

```
replace_equ :SET_empty: * - 1
        $ Set({})$ ;
        @ QED (CQFD)@
```

Likewise, the following sequence of three operations:

```
:SET:    insert  SET ; @ B0 satisfies Set(B0) (cf. 10 above)@
substitute_equ  $y = B0$ * - 1 ;
        @ the axiom SET defines the consequences@
replace_equ *-1 :separ1:;
```

could be shortened to a single one:

```
replace_equ SET, :separ1:
        $(exists x in U | x in B0) or B0 = ({})$ ;
        @ B0 satisfies Set(B0) (cf. 10 above)@
```

Based on our experience, we believe that obvious logical inference mechanisms can reduce the length of complex proof programs by at least thirty percent.

Proof Engineering

The same theory declaration may be used as the starting point for constructing proofs of several distinct theorems. For each such theorem it may be convenient to augment the original theory declaration by introducing new abbreviations, or by inserting a theorem in the theorems declaration. When submitting a proof program to the NAP proof-checker, the display resembles the example of a session transcript in Appendix B. Comments are listed in an extra column on the right of the column labeled "formula."

The examples found in the appendix illustrate some of the possibilities of engineering modular proofs in incremental steps: proof engineering is closely related to software engineering! NAP features could be used, for instance, to structure proofs in accordance with Lamport's suggestions; see [49]. See [48] for a detailed step-wise proof construction and hierarchical refinement. Criteria for a convincing proof are:

- it is simple
  - i.e., the proof follows an understandable thread, and is modular
  - the collated comments constitute an acceptable informal proof
- it is completed
  - NAP accepts a new theorem when a proof is completed
  - NAP detects partial completion when a goal is proved, and unproved working premises remain

Modularity means:

- separation of concerns
  - expell from the main proof thread all intermediate results that could be derived from other theories defined in the system
  - create independent lemmas for each derivation that is potentially re-usable

Re-usability and portability are achieved in very simple ways:

- copy-paste
  - use a labeling convention for specific proof-steps
  - use relative addressing in referring to proof-steps within argument lists
- theory update
  - update a theory "theorems" list, insert abbreviations, and declare a new theory that inherits a limited number of proved theorems.

    For instance, after proving the "empty set theorem," the theory declaration may be augmented by:

```
theorems:
 empty_set_thm esias
 ;
```

  - keep theory declarations small by removing previously established theorems that are not effectively used in a given proof

One of the keys to proof design is:

- suspending disbelief

- create a proof thread delineated by working premises (conjectures) that can trigger a partial completion detection whenever a goal or subgoal theorem is proved
- prove each working premise in the proof thread independently
- design an informal proof on paper before using NAP.

Areas of Current Research

NAP uses no resolution or tactics. More experience is needed to determine whether these features are essential for the specific goal of NAP, which is to provide a formal proof write-up after an informal one is available. Another problem concerns induction. Induction proofs are essential, but they involve lots of details. Establishing these proofs is as difficult as defining bounds in loops or recursion, and guaranteeing termination. It is not yet clear how induction can be simplified within NAP.

The main problem for future development of NAP is the development of a theory database management system that monitors,

- the consistency of a dynamically changing set of independently evolving theories, each one potentially formed by inheritance from other theories, and
- multiple conservative extensions of a theory.

6. Conclusion

It remains a utopian dream to maintain software exclusively at the abstract specification level, and to develop software solely by mapping mathematical problem specifications into efficient code by mechanically verified transformations. Although a comprehensive framework exists to fulfill this "dream" (see [41], [9], [62], [63], and [65]), the extreme benefits would be offset by even more extreme costs using existing technology. Pepper and Wirsing say that verification activities would contribute the major costs, and that these costs would make a fully formal approach infeasible with current technology. They "accept the fact that a great part of the development process has to be carried out on a more or less informal level," and a rigorous mathematical style (paper and pencil proof) is most likely to be used for proving the most difficult derivational steps.

Nevertheless, progress is being made within the academic community, and a rudimentary formal software technology has been utilized in practical settings within industry. Large projects like VDM, [6], Larch, [41], and KORSO, [63], or derived efforts like KIV, [65], have clarified issues and have provided well defined frameworks for implementing "sound" and "robust" program development systems.

Within industry semi-formal graphical representations of sets and maps are used by CASE tools to model data using the Entity-Relationship approach (see

[16]) or OMT; see [67]. They are also used to define functional specifications with Data-Flow diagrams (see [91]), and to develop real-time specifications in a synchronous system model by Statecharts; see [42]. Case tools used within these software engineering approaches also have an informal capability to transform these initial specifications. In particular, it is standard nowadays for integration testing applications to be hand-derived with machine assistance from the kinds of semi-formal specifications just described.

Our approach to formal program development is within the framework of Pepper and Wirsing, but is distinguished from their work in several ways. Our application domain is limited to the implementation of combinatorial nonnumerical algorithms. Algorithm implementation is extremely labor intensive, and is growing in importance as the demand for computer technology shifts from easier applications to more complex algorithmic ones; e.g., optimizing compilers for supercomputers, intricate data structures to implement efficient solutions to operations research problems, search and analysis algorithms in genetic engineering, complex tools for personal workstations, systems for computer-aided manufacturing, and tools for computer-aided software engineering (CASE). It is also a difficult problem that is not solved by current CASE tools and software management disciplines, which are oriented towards data processing and other applications, where the implementation and a prediction of its resource utilization follow more directly from the specification.

Our transformational methodology has been tailored to algorithms. The three basic transformation schemas (dominated convergence, finite differencing, real-time simulation) capture basic principles of algorithm design, and are guided by complexity considerations. The use of top-down refinement to transform specifications from higher to lower levels of abstraction allows us to always focus on local contexts for extracting the semantic detail needed to justify transformations. The expressive power of full set theory is used to model complex data structures and algorithmic strategies. We use a set-theoretic language to specify every abstract level of specification from the initial mathematical problem statement to the final implementation. The final implementation, which is written in terms of set-theoretic primitives, is mapped into a conventional architecture using a type system that depends only on the model of computation and that conceals the actual low level implementation language. The ability to restrict our attention to local contexts in a set-theoretic language allows us also to restrict our logic to set-theoretic semantics, and to ignore the idiosyncratic semantics of a low level language (whose semantics would be important if global program analysis were needed, see [20] and [22]).

An important objective of this work has been to overcome a serious shortcoming in the APTS transformational system. Currently, all transformations used within APTS preserve program semantics, but this is only proved on paper outside the system. Since APTS is capable of unbridled production of dangerously large quantities of code without any human intervention or oversight, the absence in APTS of machine-assisted meta-level support to prove transforma-

tions correct is a major deficiency that needs to be overcome. Our experiment with NAP indicates that this shortcoming can be overcome.

However, our experiment failed to demonstrate that this approach is practical. We have mechanically proved the hardest derivational steps. We also showed how the derivation of a high level bisimulation algorithm actually led to an improved algorithm for DFA minimization. This kind of derivational reuse saved us a great deal of labor in mechanical verification. But approximately 7,000 lines of NAP text were used to define the theory, and to work out the whole proof of identity (3.7). We believe that this amount of "proofware" represents about ten times the size of a legible mathematical proof. We would hope that a high level language approach to proof construction and reuse would help. It would also be intriguing to find out whether fast decision procedures for fragments of set theory (see [13]) could be utilized to automate and shorten our proofs.

Acknowledgements.

J. P. Keller was partially supported by ONR grant N00014-93-1-1036 and by the Italian P.F. CNR, which funded an earlier implementation of a proof construction system called Etna. R. Paige was supported by ONR grant N00014-93-1-0924, AFOSR grant AFOSR-91-0308, and NSF grant MIP-9300210. Part of this research was done while R. Paige was visiting DIKU at the University of Copenhagen.

We are grateful to Carolyn Talcott for pointing out several errors in an earlier draft of the paper. Eugenio Omodeo shared several insights that helped to clarify our thinking and improve the paper. The set of inference operations of NAP have been largely inspired by D. Cantone and A. Ferro of the University of Catania in Italy.

Appendix: A. NAP Rules and Schemes

1. The Inference Rules

rule name	*from ...*	*infer ...*
modus ponens	A, A $\rightarrow$ B	B
deny the disjunct	B $\vee$ A, $\neg$A	B
disjunctive syllogism	A $\vee$ B, A $\rightarrow$ C, B $\rightarrow$ D	C $\vee$ D
deny the consequent	B $\rightarrow$ A, $\neg$ A	$\neg$ B
hypothetical syllogism	A $\rightarrow$ B, B $\rightarrow$ C	A $\rightarrow$ C

2. The Tautological Rewriting Rules

rule name	*from ...*	*infer ...*
double negation insert	A	$\neg$ $\neg$ A
double negation drop	$\neg$ $\neg$ A	A
definition of imply	A $\rightarrow$ B $\neg$A $\vee$ B	$\neg$A $\vee$ B A $\rightarrow$ B
distribute negation over imply	$\neg$(A $\rightarrow$ B) A $\wedge\neg$B	A $\wedge\neg$B $\neg$(A $\rightarrow$ B)
contrapositive	A $\rightarrow$ B A iff B	$\neg$B $\rightarrow$ $\neg$A $\neg$A iff $\neg$B
commute equivalents commute equals commute conjuncts commute disjuncts	A_1 $\ldots op$ A_i $op \ldots$ $\ldots op$ A_j $op \ldots A_n$ if $op \in$ {iff, $=, \wedge, \vee$}	A_1 $\ldots op$ A_j $op \ldots$ $\ldots op$ A_i $op \ldots A_n$
commute different	$A_1 \neq A_2$	$A_2 \neq A_1$
law of biconditional	A iff B	(A $\rightarrow$ B) $\wedge$ (B $\rightarrow$ A)
form a conjunction	$A_1, A_2, \ldots, A_n$	$A_1 \wedge A_2 \wedge \ldots \wedge A_n$
extract a conjunct	$A_1 \wedge A_2 \wedge \ldots \wedge A_i$ $\wedge \ldots \wedge A_n$	A_i
form a disjunction (expansion rule)	A	A $\vee$ B

3. The Predicate Substitution Rules

rule name	*from . . .*	*infer . . .*
commute under quantifier	$\forall$ (resp $\exists$) . . . $x_i \in S_i \ldots x_j \in S_j$ $\ldots \mid K(x_1, \ldots, x_n)$ if $x_k (k \geq i)$ doesn't occur in S_i nor in S_j	$\forall$ (resp $\exists$) . . . $x_j \in S_j \ldots x_i \in S_i$ $\ldots \mid K(x_1, \ldots, x_n)$
quantifier negation	$\neg\forall$ $(resp\ \exists)$ $x_1 \in S_1 \ldots$ $\mid K(x_1, \ldots, x_n)$	$\neg\exists$ $(resp\ \forall)$ $x_1 \in S_1 \ldots$ $\mid \neg K(x_1, \ldots, x_n)$
universal specification	$\forall x \in S \mid K(x)$ if c is any constant	$c \in S \ \rightarrow K(c)$
	$\forall \ldots x_{i-1} \in S_{i-1},$ $x_i \in S_i, x_{i+1} \in S_{i+1},$ $\ldots x_n \in S_n \mid K(x_1, \ldots, x_n)$ if c is any constant	$\forall \ldots x_{i-1} \in S_{i-1},$ $x_{i+1} \in S_{i+1},$ $\ldots x_n \in S_n \mid c \in S_i \ \rightarrow$ $K(x_1 .. x_{i-1}, c, x_{i+1}, \ldots, x_n)$
existential specification	$\exists x \in S \mid K(x)$ if c is a Henkin constant for this existential formula	$c \in S \ \wedge \ K(c)$
universal generalization	K if x is free in K	$\forall x \in U \mid K$
	$x \in S \ \rightarrow K$ if x is free in K and doesn't occur in S	$\forall x \in S \mid K$
existential generalization	K if t is a term which occurs in K and x is a variable substitutible to t in K	$\exists x \in U \mid K(t/x)$
	$t \in S \ \wedge K$ if t is a term which occurs in K and x is a variable substitutible to t in K	$\exists x \in S \mid K(t/x)$

3. The Predicate Substitution Rules (continued)

rule name	*from . . .*	*infer . . .*
replace equivalents	A iff B, C if A is a subformula of C and all the free variables of B other than those occurring in A do not occur in C	$C(A/B)$ replace all occurrences of A by B while substituting the terms corresponding to the free variables common to A and B
replace equals	$t = u$, A if t, u are terms, x is a variable not occurring in A, t or u and if t occurs in A and u is substitutible to x in A(t/x)	$A(t/u)$
substitute equals	A if t is any term substitutible to x in A	$A(t/x)$
substitute FM	$FM(x)$ iff $aFla$, A if x is free in $aFla$ and FM occurs in A and if every free variable of $aFla$ does not occur bound in A	$A(FM/aFla)$ unify, i.e., for each occurrence of FM(u) in A substitute an instance of $aFla(x/u)$

4. The Predicate Insertion Rules

rule name	*from . . .*	*infer . . .*
let equ	if t is a variable symbol not occurring free in $aTerm$	$t = aTerm$
	if f is a new formula abbreviation and does not occur in aFla	f iff $aFla$
insert axiom	if A is a known axiom or theorem of the current theory	A

5. Specialized Set-Theoretic Rules

rule name	*from ...*	*infer ...*
extensionality	$x = t_1 \vee x = t_2 \vee .. \vee x = t_n$ $x \in \{t_1, t_2, \ldots, t_n\}$	$x \in \{t_1, t_2, \ldots, t_n\}$ $x = t_1 \vee x = t_2 \vee .. \vee x = t_n$
	$K(a_1) \wedge K(a_1) \wedge .. \wedge K(a_n)$ $K(a_1) \vee K(a_1) \vee .. \vee K(a_n)$ if x is substitutible to $a_1, a_2, \ldots, a_n$ in K	$\forall x \in \{a_1, a_2, \ldots, a_n\} \mid K(x)$ $\exists x \in \{a_1, a_2, \ldots, a_n\} \mid K(x)$
	$\forall x \in \{a_1, a_2, \ldots, a_n\} \mid K(x)$ $\exists x \in \{a_1, a_2, \ldots, a_n\} \mid K(x)$	$K(a_1) \wedge K(a_2) \wedge .. \wedge K(a_n)$ $K(a_1) \vee K(a_2) \vee .. \vee K(a_n)$

Recall that *current step rank*, denoted by $\star$, is the rank of the current derivation step, and that $rk(A)$ is the rank in a derivation of formula A, i.e., the rank of the derivation step in which A is the resulting formula.

6. The Proof Schemes

rule name	*from ...*	*infer ...*
conditional proof	A, B	A $\rightarrow$ B $premises(\star)$:= $premises(rk(A)) \cup$ $premises(rk(B))$ $less\ rk(A)\ less\ rk(B)$
working premise (postponed lemmas, conjectures, absurd hypothesis, directly evaluable assertion)	let the current resulting formula A be a premise i.e., a working hypothesis	A $premises(current\ step\ rank)$:= $\{current\ step\ rank\}$
indirect proof (proof by contradiction)	A, $\neg$ A if rk(B) $\in$ premises(rk($\neg$ A)) i.e., if B is a working hypothesis presiding to the detected contradiction	$\neg$B $premises(\star)$:= $premises(rk(A)) \cup$ $premises(rk(B))$ $less\ rk(B)$
induction proof (Peano's induction)	$A_{i_0} \wedge natural(i)$ $\wedge\ i \geqq i_0$ $\wedge A_i \rightarrow A_{i+1}$ let B be the formula : $i \geqq i_0 \wedge A_i \rightarrow A_{i+1}$ and let rk(B) be its rank	A_{i_p} $premises(\star)$:= $premises(rk(A_{i_0})) \cup$ $premises(rk(B))$ $less\ rk(A_{i_0})\ less\ rk(B)$

Appendix: B. Proof Script For The Emptyset Theorem

zf0-id card

source	symbols
FOLE	U (Func 0) notin (Pred 2) Set (Pred 1) FM (Pred 1) in (Pred 2) /= (Pred 2) = (Pred 2)

abbrev.	axioms & theorem statement
idemOr	FM(x) iff FM(x) or FM(x)
SET	Set(y) iff (exists x in U | x in y) or y = ({})
absence	x notin y iff not (x in y)
SEPARATION	Set(A) impl (exists B in U | Set(B) and (forall x in U | x in B iff x in A and FM(x)))
inequality	x /= y iff not (x = y)
equality	x = x
idemAnd	FM(x) iff FM(x) and FM(x)

premises	nb	operation	formula
	1	play: zf0 |-	forall x in U | x notin ({})
	2	insert SEPARATION	Set(A) impl (exists B in U | Set(B) and (forall x in U | x in B iff x in A and FM(x)))
	3	substitute equ A = ({}) 2	Set({}) impl (exists B in U | Set(B) and (forall x in U | x in B iff x in ({}) and FM(x)))
	4	substitute equ FM(x) iff not (x = x) 3	Set({}) impl (exists B in U | Set(B) and (forall x in U | x in B iff x in ({}) and not (x = x)))
5	5	wp	Set({})
5	6	modus ponens 4 5	exists B in U | Set(B) and (forall x in U | x in B iff x in ({}) and not (x = x))
5	7	ES B = B0 6	B0 in U and (Set(B0) and (forall x in U | x in B0 iff x in ({}) and not (x = x)))
5	8	conjunct 2 7	Set(B0) and (forall x in U | x in B0 iff x in ({}) and not

```
|    |       |    |                     |  (x = x))                      |
|    |   5   | 9  |  conjunct 2 8       | forall x in U | x in B0 iff x |
|    |       |    |                     |  in ({}) and not (x = x)       |
|    |   5   | 10 |  conjunct 1 8       | Set(B0)                        |
|    |  11   | 11 |  wp                 | exists x in U | x in B0        |
|    |  11   | 12 |  ES x = c 11        | c in U and c in B0             |
|    |  11   | 13 |  conjunct 2 12      | c in B0                        |
|    |   5   | 14 |  US x = c 9         | c in U and (c in B0 iff c in   |
|    |       |    |                     | ({}) and not (c = c))          |
|    |   5   | 15 |  conjunct 2 14      | c in B0 iff c in ({}) and not  |
|    |       |    |                     |  (c = c)                       |
|    | 11 5  | 16 |  replace equ 15 13  | c in ({}) and not (c = c)      |
|    |       | 17 |  insert equality    | x = x                          |
|    |       | 18 |  substitute equ x = | c = c                          |
|    |       |    |  c 17               |                                |
|    | 11 5  | 19 |  conjunct 2 16      | not (c = c)                    |
|    |   5   | 20 |  indir proof 11 19  | not (exists x in U | x in B0)  |
|    |       |    | 18                  |                                |
|    |       | 21 |  insert SET         | Set(y) iff (exists x in U | x  |
|    |       |    |                     |  in y) or y = ({})             |
|    |       | 22 |  substitute equ y = | Set(B0) iff (exists x in U |   |
|    |       |    |  B0 21              | x in B0) or B0 = ({})          |
|    |   5   | 23 |  replace equ 22 10  | (exists x in U | x in B0) or   |
|    |       |    |                     | B0 = ({})                      |
|    |   5   | 24 |  deny disjunct 23 2 | B0 = ({})                      |
|    |       |    | 0                   |                                |
|    |   5   | 25 |  replace equ 24 20  | not (exists x in U | x in ({}  |
|    |       |    |                     | ))                             |
|    |   5   | 26 |  Quant Neg 25       | forall x in U | not (x in ({}  |
|    |       |    |                     | ))                             |
|    |       | 27 |  insert absence     | x notin y iff not (x in y)     |
|    |       | 28 |  substitute equ y = | x notin ({}) iff not (x in ({  |
|    |       |    |  ({}) 27            | }))                            |
|    |       | 29 |  comm equiv 1 2 28  | not (x in ({})) iff x notin (  |
|    |       |    |                     | {})                            |
|    |   5   | 30 |  replace equ 29 26  | forall x in U | x notin ({})   |
--->**** you've just proved, based upon 1 yet unproved assumption(s)
the Lemma: empty_set_thm!!!
|    |       | 31 |  play zf0           | Set({})                        |
|    |       | 32 |  substitute equ y = | Set({}) iff (exists x in U |   |
|    |       |    |  ({}) 21            | x in ({})) or ({}) = ({})      |
|    |       | 33 |  substitute equ x = | ({}) = ({})                    |
|    |       |    |  ({}) 17            |                                |
|    |       | 34 |  form a disj 33 exi | ({}) = ({}) or (exists x in U  |
|    |       |    | sts x in U | x in ( |  | x in ({}))                  |
|    |       |    | {})                 |                                |
|    |       | 35 |  comm equiv 1 2 34  | (exists x in U | x in ({})) o  |
|    |       |    |                     | r ({}) = ({})                  |
|    |       | 36 |  comm equiv 1 2 32  | (exists x in U | x in ({})) o  |
```

```
|    |       |    |                      | r ({}) = ({}) iff Set({})     |
|    |       | 37 |  replace equ 36 35   | Set({})                       |
--->**** ee_is_a_set : Set({}) is proved!!!
--->**** empty_set_thm : forall x in U | x notin ({}) is proved!!!
```

Appendix: C. NAP Proof Specification For The Emptyset Theorem

The following is the theory declaration used by NAP to generate the script in Appendix B.

```
theory:
   name:zf0
   source:FOLE
   Symbols:
      U (Func 0)
      Set (Pred 1)
      notin (Pred 2)
      /= (Pred 2)
      = (Pred 2)
      in (Pred 2)
      FM (Pred 1)
      c (Symb 0)
      B0 (Symb 0)
      ;
   abbreviations:
      equality :=  (x = x);
      SEPARATION :=  (Set(A) impl (exists B in U |
      (Set(B) and
          (forall x in U |  ( (x in B) iff
               ( (x in A) and FM(x)))))));
      inequality := ( (x /= y) iff not ( (x = y)));
      empty_set_thm := forall x in U |  (x notin ({}));
      SET :=  (Set(y) iff ((exists x in U |  (x in y)) or (y = ({}))));
      esias := Set({});
      idemOr :=  (FM(x) iff  (FM(x) or FM(x)));
      absence :=  ( (x notin y) iff not ( (x in y)));
      idemAnd :=  (FM(x) iff  (FM(x) and FM(x)));
   axioms:
      inequality SEPARATION equality idemAnd absence idemOr SET
           ;
```

The following is the proof text that is used by NAP to generate the main part of the script in Appendix B.

```
theorem:
    proof: empty_set_thm
      theorem zf0 |- empty_set_thm $forall x in U |  (x notin ({}))$ ;
              @ no element is a member of {}@
      insert  $SEPARATION$ ; @ separation is the key axiom here@
      substitute_equ  $FM(x) iff not (x = x)$ * - 1 ;
              @ FM(x) is never satisfied@
      substitute_equ  $A = ({})$ * - 1 ;
              @ take an A for which Set(A) holds@
      wp $Set({})$ ; @ Set({}) is easy to verify (see below)@
      modus_ponens  * - 2 * - 1 ;
              @ B is the empty set, defined in comprehension by
              an un-satisfiable formula@
      ES  $B = B0$ * - 1 ; @ let B0 be an object like B@
      conjunct  2, * - 1 ;
    :separ1:  conjunct  1, * - 1 ;@ Set(B0) @
    :separ2:  conjunct  2, * - 2 ;@ every element of B0 satisfies FM@
    :absurd:  wp $exists x in U |  (x in B0)$ ;
              @ let's assume B0 has an element@
      ES  $x = c$ * - 1 ; @ let it be c@
      conjunct  2, * - 1 ;
      US  $x = c$ :separ2: ;
      statement $ (c in U)$ ;
      modus_ponens  * - 2 * - 1 ;
      replace_equ *-1 *-4 ;
              @ drop c in B0 to get c in ({}) and not (c = c)@
    :eq:  insert  $equality$ ;
              @ c satisfies FM, whence violates equality...@
      substitute_equ  $x = c$ * - 1 ;
      conjunct  2, * - 3 ; @ contradiction with equality@
    :non_absurd:      indir_proof  :absurd: * - 1 * - 2 ;
              @ B0 may not have a member!@
    :SET:  insert  $SET$ ; @ B0 satisfies Set(B0) (cf. 9 above)@
      substitute_equ  $y = B0$ * - 1 ;
              @ the axiom SET defines the consequences@
      replace_equ *-1 :separ1:
              $(exists x in U | x in B0) or B0 = ({})$ ;
      deny_disjunct  * - 1 :non_absurd: ;
              @ B (i.e., its instance B0) is really the empty set
              (cf. 6 above)@
      replace_equ *-1 :non_absurd:;
              @ and *-2 above says explicitly! there is no element in B0@
      Quant_Neg  * - 1 ;
    :almost:  dbl_negation_drop  * - 1 ;
      insert  $absence$ ;
              @ the predicate notin is defined by the 'absence' axiom@
      substitute_equ  $y = ({})$ * - 1 ;
      comm_iff_  1 2 ,* - 1 ;
```

```
  replace_equ * - 1 :almost: ; @almost QED@

  play zf0 |- esias $Set({})$ ;
          @ esias stands for: e-mpty s-et i-s a- s-et @
:SET_empty:  substitute_equ  $y = ({})$ :SET: ;
  substitute_equ  $x = ({})$ :eq: ;
          @ equality: the 1st term of a disjunct@
  form_a_disj  * - 1 $exists x in U | x in ({})$ ;
          @ the 2nd term may be anything, since the 1st term holds@
  comm_or_  1 2 ,* - 1 ;
  comm_iff_  1 2 :SET_empty: ;
  replace_equ * - 1 * - 2;
          @ QED (CQFD)@
```

Bibliography

[1] ADA UK News, Vol. 6, No. 1, January 1985, pp. 14–15.

[2] Aho, A., Hopcroft, J., and Ullman, J., *Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Massachusetts, 1974.

[3] Bauer, F., and Wossner, H., *Algorithmic Language and Program Development,* Springer-Verlag, Berlin, 1982.

[4] Bell, J. L., and Slomson, A. B., *Models and Ultraproducts,* North-Holland, Amsterdam, 1969.

[5] Biersack, M., Raschke, R., and Simons, M., *The DevaWEB System,* Technical Report 93-39, Technische Universität Berlin, December 1993.

[6] Bjorner, D., and Jones, C., *VDM'87,* Springer-Verlag, Berlin, 1987.

[7] Bloom, B., and Paige, R., *Transformational design and implementation of a new efficient solution to the ready simulation problem,* Sci. Comput. Programming 24, No. 3, 1995, pp. 189–220.

[8] Boyer, R., and Moore, J., *A Computational Logic,* Academic Press, New York, 1979.

[9] Broy, M., *Experiences with Software Specification and Verification Using LP, the Larch Proof Assistant,* Research Report 93, DEC Systems Research Center, Palo Alto, California, 1992.

[10] Burstall, R., and Darlington, J., *A transformation system for developing recursive programs,* J. ACM 24, No. 1, January 1977, pp. 44–67.

[11] Cai, J., Facon, P., Henglein, F., Paige, R., and Schonberg, E., *Type transformation and data structure choice,* pp. 126–124 in: *Constructing Programs From Specifications,* B. Moeller, ed., North-Holland, Amsterdam, 1991.

[12] Cai, J., and Paige, R., *Program derivation by fixed point computation,* Sci. Comput. Programming 11, No. 3, 1988/1989, pp. 197–261.

[13] Cantone, D., Ferro, A., and Omodeo, E., *Computable Set Theory,* Clarendon Press, Oxford, 1989.

[14] Cardon, A., and Crochemore, M., *Partitioning a graph in* $o(|a|log|v|)$, Theoret. Comput. Sci. 19, 1982, pp. 85–98.

[15] Cheatham, T., and Wegbreit, B., *A laboratory for the study of automating programming,* pp. 208–211 in: *Proceedings AFIPS 1972 Spring Joint Computer Conference,* 1972.

[16] Chen, P., *The entity-relationship model — toward a unified view of data,* ACM TODS 1, No. 1, 1976.

[17] Cocke, J., and Kennedy, K., *An algorithm for reduction of operator strength,* CACM 20, No. 11, November 1977, pp. 850–856.

[18] Cocke, J., and Schwartz, J., *Programming Languages and Their Compilers,* Lecture Notes, Courant Institute, New York University, New York, 1969.

[19] Coquand, T., and Huet, G., *The calculus of constructions,* Inform. and Comput. 76, 1988, pp. 95–120.

[20] Cousot, P., and Cousot, R., *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points,* pp. 238–252 in: *Proceedings Fourth ACM Symposium on Principles of Programming Languages,* 1977.

[21] Cousot, P., and Cousot, R., *Constructive versions of Tarski's fixed point theorems,* Pacific J. Math. 82, No. 1, 1979, pp. 43–57.

[22] Cousot, P., and Cousot, R., *Semantic foundations of program analysis,* pp. 303–342 in: *Program Flow Analysis,* S. Muchnick and N. Jones, eds., Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[23] Davis, M., *Obvious logical inferences,* pp. 530–531 in: *Proceedings of IJCAI 81,* 1981.

[24] Davis, M., and Schwartz, J., *Metamathematical extensibility for theorem verifiers and proof checkers,* Comp. Math. Appl. 5, 1979, pp. 217–230.

[25] Deak, E., *A Transformational Approach to the Development and Verification of Programs in a Very High Level Language,* Courant Computer Science Report 22, Courant Institute, New York University, New York, November 1980.

[26] Despeyroux, T., *TYPOL a Formalism to Implement Natural Semantics,* Research Report, INRIA, Sophia Antipolis, France, 1986.

[27] Dijkstra, E., *A Discipline of Programming,* Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[28] Dijkstra, E. W., *A constructive approach to the problem of program correctness,* BIT 8, No. 3, 1968, pp. 174–186.

[29] Doberkat, E. E., and Gutenbeil, U., *SETL to Ada tree transformations applied,* Inform. Software Tech. 29, No. 10, December 1987, pp. 548–557.

[30] Donzeau-Gouge, V., Dubois, C., Facon, P., and Jean, F., *Development of a programming environment for SETL,* pp. 21–32 in: *Proceedings ESEC 87,* H. Nichols and D. Simpson, eds., LNCS Vol. 289, Springer-Verlag, Berlin, 1987.

[31] Donzeau-Gouge, V., Kahn, G., Lang, B., Melese, B., and Morcos, E., *Outline of a tool for document manipulation,* pp. 615–620 in: *IFIP '83,* North-Holland Elsevier, Amsterdam, 1983.

[32] Dubois, C., *Determination statique des types pour le langage SETL,* Doctoral Dissertation, CNAM, Paris, France, 1989.

[33] Earley, J., *High level iterators and a method for automatically designing data structure representation,* J. Comput. Languages 1, No. 4, 1976, pp. 321–342.

[34] Fernandez, J.-C., *An implementation of an efficient algorithm for bisimulation equivalence,* Sci. Comput. Programming 13, 1989/90, pp. 219–236.

[35] Fong, A., and Ullman, J., *Induction variables in very high level languages,* pp. 104–112 in: *Proceedings Third ACM Symposium on Principles of Programming Languages,* January 1976.

[36] Freudenberger, S., Schwartz, J., and Sharir, M., *Experience with the SETL optimizer,* ACM TOPLAS 5, No. 1, 1983, pp. 26–45.

[37] Garland, S., and Guttag, J., *A Guide to LP, the Larch Prover,* Research Report 82, DEC Systems Research Center, Palo Alto, California, 1991.

[38] Gordon, M., and Melham, T., eds., *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic,* Cambridge University Press, 1993.

[39] Gries, D., *Describing an algorithm by Hopcroft,* Acta Inform. 2, 1973, pp. 97–109.

[40] Gries, D., *The Science of Programming,* Springer-Verlag, New York, 1981.

[41] Guttag, J., and Horning, J., *Larch: Languages and Tools for Formal Specification,* Springer-Verlag, New York, 1993, with S. Garland, K. Jones, A. Modet, and J. Wing.

[42] Harel, D., *On visual formalisms,* CACM 31, No. 5, 1988, pp. 514–530.

[43] Hoare, C. A. R., *An axiomatic basis for computer programming,* CACM 12, No. 10, 1969, pp. 576–580.

[44] Hopcroft, J., *An n log n algorithm for minimizing states in a finite automaton,* pp. 189–196 in: *Theory of Machines and Computations,* Z. Kohavi and A. Paz, eds., Proceedings International Symposium on Theory of Machines and Computation, Academic Press, New York, 1971.

[45] Kahn, G., *Natural Semantics,* Research Report, INRIA, Sophia Antipolis, France, 1987.

[46] Kanellakis, P., and Smolka, S., *CCS expressions, finite state processes and three problems of equivalence,* Inform. and Comput. 86, No. 1, 1990, pp. 43–68.

[47] Keller, J., *Cantor: A Tutorial and a User's Guide,* Education Report 94/9, Kepler, Paris, 1994.

[48] Keller, J., and Paige, R., *Algorithm Design and Verifiable APTS Transformational Derivations with NAP,* Research Report 94/11, Kepler, Paris, 1994.

[49] Lamport, L., *How to Write a Proof,* Technical Report 94, DEC Systems Research Center, Cambridge, Massachusetts, February 1993.

[50] Milner, R., *A Calculus of Communicating Systems,* Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, Berlin, 1980.

[51] Milner, R., *Calculi for synchrony and asynchrony,* Theoret. Comput. Sci. 25, No. 3, 1983, pp. 267–310.

[52] Myhill, J., *Finite Automata and the Representation of Events,* WADD TR-57-624, Wright Patterson AFB, Ohio, 1957.

[53] Nerode, A., *Linear automaton transformations,* Proc. Amer. Math. Soc. 9, 1958, pp. 541–544.

[54] Omodeo, E., 1994, private communication.

[55] Paige, R., *Real-time simulation of a set machine on a ram,* pp. 69–73 in: *Computing and Information,* N. Janicki and W. Koczkodaj, eds., Vol. II, Canadian Scholars' Press, Toronto, May 1989.

[56] Paige, R., *Efficient translation of external input in a dynamically typed language,* pp. 603–608 in: *Technology and Foundations — Information Processing 94,* B. Pehrson and I. Simon, eds., Vol. 1 of IFIP Transactions A-51, Conference Record of IFIP Congress 94, North-Holland, Amsterdam, September 1994.

[57] Paige, R., *Viewing a program transformation system at work,* pp. 5–24 in: *Programming Language Implementation and Logic,* M. Hermenegildo and J. Penjam, eds., LNCS Vol. 844, Springer-Verlag, Berlin, September 1994, Proceedings Joint 6th International Conference on Programming Language Implementation and Logic Programming (PLILP) and 4th International Conference on Algebraic and Logic Programming (ALP).

[58] Paige, R., and Koenig, S., *Finite differencing of computable expressions,* ACM Trans. Programming Languages and Systems 4, No. 3, 1982, pp. 401–454.

[59] Paige, R., and Schwartz, J., *Expression continuity and the formal differentiation of algorithms,* pp. 58–71 in: *Proceedings 4th ACM Symposium on Principles of Programming Languages,* January 1977.

[60] Paige, R., and Tarjan, R., *Three partition refinement algorithms,* SIAM J. Comput. 16, No. 6, 1987, pp. 973–989.

[61] Paulson, L., *Isabelle: The next 700 theorem provers,* pp. 361–386 in: *Logic and Computer Science,* P. Odifreddi, ed., Academic Press, New York, 1990.

[62] Pepper, P., *A simple calculus for program transformation,* Sci. Comput. Programming 9, 1987, pp. 221–262.

[63] Pepper, P., and Wirsing, M., *Korso: A Methodology for the Development of Correct Software,* Technical Report 93-36, Technische Universität Berlin, November 1994.

[64] Reif, W., *Formal software development in the KIV system,* in: *KORSO: Methods, Languages, and Tools for the Construction of Correct Software,* M. Broy and S. Jähnichen, eds., LNCS Vol. 1009, Springer-Verlag, Berlin, 1995.

[65] Reif, W., *The KIV-approach to software verification,* in: *KORSO: Methods, Languages, and Tools for the Construction of Correct Software,* M. Broy and S. Jähnichen, eds.,

LNCS Vol. 1009, Springer-Verlag, Berlin, 1995.

[66] Reif, W., Schellhorn, G., and Stenzel, K., *Interactive correctness proofs for software modules using KIV,* pp. 151–162 in: *Compass 95, Proceedings of the 10th Annual Conference on Computer Assurance,* 1995.

[67] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., eds., *Object-Oriented Modeling and Design,* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[68] Sands, D., *Total correctness by local improvement in program transformation,* pp. 221–232 in: *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages,* January 1995.

[69] Schonberg, E., Schwartz, J., and Sharir, M., *An automatic technique for selection of data representations in SETL programs,* ACM TOPLAS 3, No. 2, April 1981, pp. 126–143.

[70] Schwartz, J., *On Programming: An Interim Report on the SETL Project, Installments I and II,* New York University, New York, 1974.

[71] Schwartz, J., *Automatic data structure choice in a language of very high level,* CACM 18, No. 12, December 1975, pp. 722–728.

[72] Schwartz, J., *Optimization of very high level languages, Parts I, II,* J. Comput. Languages 1, No. 2–3, 1975, pp. 161–218.

[73] Schwartz, J., *On correct program technology,* pp. 120–146 in: *Two Papers on Program Verification,* Courant Computer Science Report No. 12, Courant Institute, New York University, New York, September 1977.

[74] Schwartz, J., Dewar, R., Dubinsky, E., and Schonberg, E., *Programming with Sets: An Introduction to SETL,* Springer-Verlag, New York, 1986.

[75] Sharir, M., *Some observations on formal differentiation,* ACM TOPLAS 4, No. 2, March 1982, pp. 000–000.

[76] Shields, D., *Measuring SETL Performance,* Doctoral Dissertation, Computer Science Department, New York University, 1983.

[77] Shoenfield, J., *Mathematical Logic,* Addison-Wesley, Reading, Massachusetts, 1967.

[78] Sintzoff, M., Weber, M., de Groote, P., and Cazin, J., *Definition 1.1 of the generic development language Deva,* Tool Use Research Report 94, Unité d'Informatique, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 1989.

[79] Smith, D., *Kids — a knowledge-based software development system,* pp. 129–136 in: *Proceedings Workshop on Automating Software Design, AAAI-88,* September 1988.

[80] Snyder, K., *The SETL2 Programming Language,* Technical Report 490, Courant Insititute, New York University, New York, 1990.

[81] Standish, T., Kibler, D., and Neighbors, J., *The Irvine Program Transformation Catalogue,* Technical Report, University of California at Irvine, Department of Information and Computer Science, Irvine, California, January 1976.

[82] Straub, R., *Taliere: An Interactive System for Data Structuring SETL Programs,* Doctoral Dissertation, Department of Computer Science, New York University, 1988.

[83] Suppes, P., and Sheehan, J., *CAI course in axiomatic set theory,* pp. 3–80 in: *University Level Computer-Assisted Instruction at Stanford: 1968-1980,* P. Suppes, ed., Stanford University, Institute for Mathematical Studies in the Social Sciences, Stanford, California, 1981.

[84] Tarjan, R., *Testing flow graph reducibility,* J. Comput. System Sci. 9, No. 3, 1974, pp. 355–365.

[85] Tarski, A., *A lattice-theoretical fixpoint theorem and its application,* Pacific J. Math. 5, 1955, pp. 285–309.

[86] Tenenbaum, A., *Type Determination for Very High Level Languages,* Courant Computer Science Report 3, Courant Institute, New York University, New York, 1974.

[87] Watson, B., *A Taxonomy of Finite Automata Minimization Algorithms,* Computing Science Note 93/44, Eindhoven University of Technology, Eindhoven, The Netherlands, 1993.

[88] Weber, M., Simons, M., and Lafontaine, C., *The Generic Development Language Deva,* LNCS Vol. 738, Springer-Verlag, Berlin, 1993.

[89] Westin, S., *Fast Decision of Strong Bisimulation Equivalence Using Partition Refine-*

ment, Dissertation for the Licentiate Degree in Computer Science, Chalmers University, Department of Computer Science, Göteborg, Sweden, 1989.

[90] Weyrauch, R., *A User Manual for FOL,* Technical Report 235.1, Stanford AIM, Stanford, California, 1977.

[91] Yourdon, E., *Modern Structured Analysis,* Yourdon Press, New York, 1989.

J. P. Keller
Kepler
8, rue des Haies
F-75020 Paris
France
E-mail: keller@cnam.fr

R. Paige
Department of Computer Science
Courant Institute
251 Mercer Street
New York, New York 10012
E-mail: paige@cs.nyu.edu

Received February 1995.