Compiler Implementation of ADTs Using Profile Data

A. Dain Samples*

Department of Electrical and Computer Engineering, University of Cincinnati, Cincinnati, OH 45221-0030 USA, Dain.Samples@uc.edu

Abstract. There are many possible implementations of some very useful programming abstractions (sets, lists, and maps, to name a few), and selecting from among them is currently one of the early tasks in the design of a software system. While programming discipline and/or language features may allow the user to change implementations of an abstraction relatively easily, there remains the inherent problem of selecting a consistent and efficient set of implementations for a particular program. A small set of extensions to existing languages allows the specification of the necessary profile data within that of the implementation of the abstraction. The TYPESETTER system selects a consistent and efficient set of implementations for a program's abstractions based on the collected profile data.

1 Introduction

> The 'ideal system of the future' will keep profiles associated with source programs, using the frequency counts in virtually all phases of a program's life. . . . An optimizing compiler can also make very effective use of the profile, since it often suffices to do time-consuming optimization on only one-tenth or one-twentieth of a program. (Knuth 1971)

Software construction systems of the future (including compilers) will be collecting and using much more information about software than just the source code; profile data will certainly be one such source of information. TYPESETTER was developed to explore how compilers and language systems might use profile data in the construction of software systems.

We describe how compilers can use profile data to select appropriate implementations for a program. Our approach extends an existing language (C++) to allow the specification of programming abstractions, their implementations, and relevant profile data. An earlier paper gave the broad outlines of the goals of this research (Samples and Hilfinger 1990). As in that paper, we distinguish between two different programmers that would use TYPESETTER: the *User* is writing application programs with TYPESETTER; the *Implementor* adds functionality to TYPESETTER.

* Supported in part by an AT&T Bell Laboratories Scholarship, and by Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292 at University of California, Berkeley.

2 TypeSetter: The System

The term 'software crisis' has been around for a long time, and yet programming environments still require programmers to be responsible for far too much of the implementation of a software system. While research on, say, compiler optimizations produce linear speedups for compiled code (usually on the order of 10-20% when they work), it doesn't address the problem of programming abstraction implementation or software re-use. But before programmers will make use of libraries of reusable components two goals must be met:

- Components matching the programmers' needs must be easily (i.e., automatically) found within the library. Programmers avoid looking for reusable code if the work required to find what they're looking for appears equivalent to the work of simply writing their own version of the code.
- Programmers must be satisfied that the components being used in their program are not the cause of any performance problems. If programmers too often find that the only way of speeding up their program is to rewrite someone else's code, then, again, the effort may appear equivalent to simply (re)writing their own version.

TypeSetter addresses both of these issues. The system automatically selects implementations for the components of a program, and selects the ones that offer the best performance for that particular program. In addition, TypeSetter has several other satisfying properties: (1) the User does not have to be an expert in compilers or program performance analysis to use the system; (2) the Implementor does not have to be an expert in compilers or code generation, and is required only to have some knowledge of how to characterize the performance of a function; (3) adding an alternative implementation for an abstraction is independent of previously existing implementations of that abstraction; and (4) the implementation selection algorithm used by the system is independent of the specific alternative implementations available to it. The net result is a system that addresses simultaneously the issues of code reusability, and efficient automatic construction of software systems.

Our exploration of the idea of using profile data in a compiler is based on three hypotheses. The first is that most programs obey the "90-10" rule, a widely accepted rule-of-thumb that says that 90% of execution resources are consumed by 10% of a program. The name is arbitrary, since actual numbers vary from program to program. Secondly, implementation selection can be done very quickly with a minimum of backtracking or use of complicated algorithms; furthermore, decisions based on the heavily-used 10% will almost never have to be rescinded due to information contained in the remaining 90%. And thirdly, the specification of alternative implementations, complete with profiling specification and evaluation, can be done as part of the implementation of an abstract data type (ADT), and the evaluation of the profile data can be relatively system-independent.

TypeSetter provides means for the Implementor to specify the collection and evaluation of profile data about a program's use of an ADT. An ADT-independent heuristic evaluates the profile data to select efficient implementations for the variables and functions in the User's program.

An ADT is defined as a set of functions, and may have many implementations defined for it. Each ADT has a distinguished implementation that serves not only as the default implementation but is also the implementation that collects profile data. To add an implementation for an ADT, the Implementor must implement each function in the specification of the abstraction. For each interface function, the Implementor must also write an evaluation function that estimates the resources required by that function. Finally, the Implementor may have to modify the profiling implementation of this ADT so that the necessary information for compile-time evaluation of this implementation is collected.

For each ADT implementation, the Implementor writes a feasibility function that determines whether this implementation can be assigned to a specific variable in the program being compiled, and an instantiation function that provides mechanical details of the implementation of the variable. An implementation assignment for a variable is said to be feasible when all information necessary for the implementation is available and satisfies the constraints of the implementation.

Each of these functions is discussed in more detail below.

3 TypeSetter: The Language

The TYPESETTER prototype has been implemented as a preprocessor that emits C++ code.[2] From the User's point of view, the enhancements to C++ are minimal. Figure 1 shows some variable declarations that might appear in a User's TYPESETTER program. The declarations can be minimal or can contain 'hints' which TYPESETTER can use to select more efficient implementations. For instance, the integer elements of the set *si* will never be less than zero or greater than 1024. This may allow TYPESETTER to assign *si* a more efficient fixed-size bit-array implementation. Before the User added this information to the declaration, the selected implementation would have had to be able to handle sets of unbounded size, and any implementation requiring such size information would not have been a feasible implementation assignment for that variable.

Each ADT in the system defines its own set of auxiliary declarations, some of which are required in every variable declaration, others of which are optional. These declarations provide additional information such as mapping functions or range bounds that cannot be (easily) deduced from the source code. The declaration of optional information does not constrain TYPESETTER to implementations that use the information: the most efficient set of implementations will be chosen for a program, whether or not all of the declared information is used by the resulting program.

Most of the language enhancements introduced by TYPESETTER impact the Implementor, whose task is three-fold: implement the abstraction, provide the profiling specifications, and write the evaluation functions. The first task is straightforward and requires no enhancements to the base language.

[2] The syntax used in the examples is an idealized fiction; the actual syntax used in the current prototype is less readable.

```
typedef ... MyType;
typedef ... Token;
typedef ... Value;

// simple declarations
Set(int)              a;
Set(MyType)           b;
Set(Token)            c;
Map(Token, Value)     d;

// declarations with auxiliary information
Set(int(lowerb=0, upperb=1024))                    si;
Set(MyType(objToInt=f1(), intToObj=f2(int;lowerb=-32, upperb=32))
Set(Token(order=f3()))                             stkn;
Map(Token(order=f3()), Value(lowerb=0x0000, upperb=0xFFFF))
```

Fig. 1. Auxiliary declarations

Profiling implementation: Each ADT has a distinguished implementation that is the default implementation for the abstraction, and the implementation that collects the profile data. This implementation must be sufficiently general to allow the implementation of all functions in the interface of the ADT. Figure 2 shows the code for the profiling implementation of the add-an-element function in the interface for sets.[3] This implementation, called *Set_P*, uses a linked list to insure that all functions in the interface can be implemented and profiled.

Profiling variables (declared as **profilers** in Fig. 2) are allocated *per call site* in the User's program. If the User's program calls *add* from three distinct sites, then a total of three instances each of *pcnt*, *psizeSum*, and *pwasIn* are allocated. On each call of the *add* function, the invocation counter *pcnt* is incremented, and the *psizeSum* profile variable is incremented by the current size of the set. One implementation for sets wants to know how many times *add* was invoked to add an element that was already a member: the profiling variable *pwasIn* computes that statistic.

Evaluation functions: Each implementation of a function in the interface of an ADT must have a corresponding evaluation function written for it. An evaluation function for the alternative implementations (*not* the profiling implementation) returns an estimate of the runtime resources required by the invocation of that function at a particular call site in the User's program. The profiling implementation's evaluation functions return a rough estimate of the relative importance of a particular call site in the User's program. The distinction between these uses of the evaluation functions is discussed further in section 4. Fig. 3 shows an implementation of the add-an-element function when the basic representation of the set is a bit array; the class name is *Set_bm*.

[3] This is a small example, and is not purported to be complete, or even useful as it is.

```
function Set_P::add(any e)
{
   profiler pcnt, psizeSum, pwasIn;
   Link lp;
   pcnt++; psizeSum += length;
   lp = first;
   while (lp != nil && e != lp→data) {
      lp = lp→next;
      }
   if (lp == nil) {
      // e not in the set
      Link newp = new Link;
      newp→data = e; newp→next = first;
      first = newp;
      }
   else {
      pwasIn++;
      }
}

Evaluate Set_P::add(CallSite c)
{ return psizeSum + pcnt − pwasIn; }
```

Fig. 2. Profiling implementations of *add*

Feasibility function: Finally, the Implementor must supply a *feasibility* function whose task is to evaluate whether a specific implementation can be used for a particular User's variable. This is as close as the Implementor gets to the internals of the compiler: she has to be familiar with the data structure representing (a portion of) the compiler's knowledge of the variable. The feasibility function returns *true* if this implementation can be used to implement the variable, and bases this decision on the information passed to it by the compiler. Figure 4 shows a (simplified) feasibility function for sets implemented as unbounded bit maps.

Instantiation function: Once an implementation for a variable has been selected, then the necessary source code for the implementation may need to be generated. The compiler calls the implementation's *instantiation* function which returns three sets of specifications: how the implementation is to be generated (if necessary), the coercion class necessary to maintain strong type checking in the User's code, and the variable declaration. An example of an Instantiation function written by an Implementor would not be very instructive since much of its job is simply to implement generic classes[4]. However, the generation of code to handle generic types is under the control of the Implementor.

For each user variable declared to be, say, a set of some user-type, *UType*, a naive

[4] Newer C++ compilers that implement templates (Ellis and Stroustrup 1990) will simplify Instantiation functions considerably.

```
vois Set_bm::add(any e)
{
   int i = mapToInt(e);
   int w = (i / (sizeof(int)*sizeof(byte)));
   int b = (i mod (sizeof(int)*sizeof(byte)));
   setbits[w] |= (1 << b);
}

Evaluate Set_bm::add(CallSite c)
{
   return c.pcnt *
     (idividePwr2_op + modPwr2_op +
      orAssign_op + array_op + shift_op);
}
```

Fig. 3. An alternative implementation of *add* with evaluation function

```
Feasible Set_bm(Uservar uvar)
{
   if (uvar.mapToInt.defined
       && uvar.lowerb.defined
       && uvar.upperb.defined) return true;
   else return false;
}
```

Fig. 4. Feasibility function for *Set_bm*

implementation of the generic specification of sets would create a new copy of the implementation code for sets with all instances of the generic parameter replaced with *UType*. In practice, this is often unnecessary. For example, code can be written once to handle sets of pointers to objects. However, it is important not to give up strong type checking to gain this savings in code space. Users should still be notified when their programs violate the declarations they themselves have made. All that is needed is a single implementation of sets of pointers with appropriate *coercion types* to enforce type checking on the base types. Using TYPESETTER the Implementor can generate one implementation of the set functions capable of handling pointers and coercion classes for maintaining strong type checking.

4 The Implementation Assignment Algorithm

The assignment algorithm is described in detail in the author's dissertation (Samples 1991). Here we will concentrate on communicating the basic idea. Baldly stated, the call sites are sorted by importance, and the cheapest implementation for each function is assigned in decreasing order of importance until a consistent assignment

of implementations has been found for the program. This single sentence glosses a great deal of detail, of course.

In TYPESETTER, profile data is collected per call site. While future work with TYPESETTER will look at what additional information can be gleaned from collecting profile data per variable or per object, our intuition led us to believe that the selection algorithm should work approximately in the same way that a human programmer optimizes a program based on profile data. A programmer asks "Where are the hot-spots in the program?", and "What can I do to improve the code at that location?" Answering these questions requires quantitative information about the behavior of the program at that location; in TYPESETTER such information is summarized by per-call-site profile data. In contrast with previous work in this area, TYPESETTER's assignment algorithm does no control flow analysis.

The compiler first ranks all call sites based on the value returned by an initial estimate function that predicts the potential impact of a call site on the final behavior of the program. These initial estimate functions are actually written as evaluation functions for the profiling implementation of the function (see Fig. 2). Sorting call sites solely by their execution frequency will not work: a sort function that is called once and that is $O(n \log n)$ in the size of the number of elements can easily overwhelm a function that is called n times and whose execution is $O(1)$.

The algorithm recurses down this ranked list, assigning the cheapest available implementation to the function at each call site. For a particular call site, a set of feasible implementations of the function at that call site is computed, such that each implementation in the set is consistent with all previously assigned call sites. 'Cheapest' is determined by the evaluation functions associated with each possible implementation function; they return an estimate of the runtime resources that would be required by this call site if the associated implementation were assigned to it. (Our prototype concentrates on runtime performance, and ignores space usage. See Low (Low 1974), Ramirez (Ramirez 1980) and Rowe (Rowe 1976) for discussions of metrics that incorporate both space and time.)

It is possible that the set of feasible functions for a call site is empty because either (a) there are no feasible implementations consistent with previous assignments, or (b) all of the feasible implementations have been tried without success. In either case, we say the assignment is *blocked* and the algorithm must backtrack to the previous call site in the list, unassign it and try the next cheapest implementation. If every function in the interface of an ADT had an implementation with a type signature for every combination of possible implementations, then there would be no blocking. A more realistic approach might be for TYPESETTER to generate the necessary function with an appropriate signature, as Rowe demonstrated in his work (Rowe 1976). This is not currently implemented in TYPESETTER.

There are two issues with regard to backtracking: the potentially exponential nature of the heuristic, and the performance degradation of the constructed software. The empirical results indicate that backtracking can be controlled, does not increase the running time of the compiler significantly, and the implemented programs are efficient.

In practice, the implementation assignment algorithm does very little backtracking: it zeros in on a consistent implementation rather quickly. The assignment heuristic was parameterized to force it to enumerate across all consistent implementations

and choose the one that represents the most efficient solution. This enumeration is controlled by a parameter p, $0 \leq p \leq 1$ to specify that those call sites that account for an estimated p% of the runtime resources are to be exhaustively enumerated to find the best possible implementations for those sites. By setting $p = 1$, all implementations are enumerated and the ones with the minimum estimate of cost are selected. At the other extreme, the first consistent implementation is quickly returned by setting the parameter $p = 0$. If the User's program satisfies the 90-10 rule, then setting $p = .9$ would result in a complete enumeration of all possible implementations for (approximately) 10% of the call sites, with the remaining 90% assigned the first consistent implementation found.

More precisely, the set of call sites is sorted in decreasing order of the values returned by the profiling implementation's evaluation functions. Let $S = \sum_i C_i$, where C_i is the initial estimate returned for the function at the i^{th} location in the sorted list. The sum of these values, S, is multiplied by the parameter p to determine a cutoff point k in the list of sorted call sites. The cutoff point is the smallest index k such that $\sum_{i<k} C_i >= p * S$. At each point in the assignment algorithm, if call site i is below the cutoff point in the list ($i \geq k$), only the first consistent implementation is assigned, and all others are ignored. If the call site is above the cutoff point ($i < k$), then each consistent implementation for that call site is evaluated.

5 Empirical Results

The TypeSetter prototype has nine implementations of three ADTs: Set, List, and Map. There are five implementations of Sets: *Set_P*, the profiling implementation; *Set_bmarr*, a bit-mapped implementation implemented as an array of 32-bit words; *Set_bmwrd*, a bit-mapped implementation that uses only one 32-bit word; *Set_slist*, a simple linked list; and *Set_slistord*, a linked-list implementation that keeps the contained objects sorted in the order of their memory addresses. There are two implementations each for Maps and Lists. We will limit the discussion to showing how TypeSetter performs on variables declared to be sets of User-defined objects.

There are two distinct issues that must be examined when evaluating TypeSetter. First, we want to test our hypothesis that a greedy assignment algorithm works well. We want to know how quickly an initial assignment of implementations is made, and how close that assignment is to the 'optimal' solution, assuming that the performance estimates returned by the evaluation functions are accurate.

The second issue is the accuracy of the estimates returned by the evaluation functions; i.e., how closely the final performance of the implemented program correlates with the predictions made by the Implementor's evaluation functions.

K-S: Our first example program is an implementation of Knuth and Stevenson's algorithm (Knuth and Stevenson 1973) for instrumenting a program flow graph (PFG) with profiling counters; we'll call it K-S. The algorithm finds a minimal set of nodes that are to be instrumented, and from which the execution counts of all nodes can be computed. Our implementation of K-S uses three sets: the variable *Graph* is the set of all graph objects, both nodes and arcs. Associated with each node in the graph are two sets: *gozintas*, the set of all arcs that come into the node, and *gozoutas*, the set of all arcs that exit the node.

Profile data was generated by running two small PFGs through K-S, one a small five-node graph that Knuth and Stevenson used as an example in their paper (Knuth and Stevenson 1973), and the other a six-node, 12-arc PFG. Based on that profile data and with $p = 0$, TYPESETTER selected *Set_bmarr* for the variable *Graph*, and *Set_slist* for the two arc sets, *gozintas* and *gozoutas*. This appears to be a reasonable assignment of implementations. Since *Graph* is a completely full set, there are no penalties to pay in a bitmap implementation for having to check bits in the bit vector that aren't set. This is not the case for the *gozintas* and *gozoutas* variables: the number of arcs coming into or leaving an arc is never more than three in our example graphs; a linked list would be much better for these two variables. This apparently good choice is further confirmed by running TYPESETTER with $p = 1$. After a full enumeration of the possible implementations, TYPESETTER makes exactly the same choices as it did with $p = 0$; this lends credence to our hypothesis that a greedy implementation assignment algorithm is already fairly close to 'optimal'. Table 1 are K-S's running times when the variables are assigned as shown. The input data is a 364-node PFG. The first entry in the table reflects the implementations chosen by THERBLIG, and the remainder show that it was indeed a reasonable set of implementations.

K-S runtimes		
Graph	gozintas & gozoutas	time
Set_bmarr	Set_slist	2.50s
	Set_slist	2.92s
	Set_slistord	3.37s
	Set_bmarr	3.79s

THERBLIG runtimes		
1	$p = 0$	34.92s
2	$p = .9$	32.60s
3	$p = 1$	35.98s
4	Set_slist	36.21s
5	Set_slistord	36.36s
6	Set_bmarr	152.17s
7	profiling	44.73s

Table 1. Running times

THERBLIG: THERBLIG[5] is the implementation of the assignment heuristic for the TYPESETTER system. From the descriptions of the available abstractions and their implementations, and the description of the User's program, it selects implementations for the variables declared, and functions invoked, in the User's program. THERBLIG consists of over 8500 lines of TYPESETTER code and comments. This includes almost 2500 lines of TYPESETTER code for the analysis portion of the software, with the other 6000 lines taken up by the nine implementations of the three abstractions of Sets, Lists, and Maps. There are 23 variables utilizing these abstractions: four are Lists, seven are Maps, and eleven are Sets. We concentrate on how TYPESETTER chose to implement the Set variables.

Seven different implementations of THERBLIG were compiled, either by THERBLIG itself, or by hand; the results of running the versions of THERBLIG produced by

[5] The name is based on Frank Gilbreth's unit of time-motion(Gilbreth, Jr. and Carey 1948).

compiling each of the implementations is in Table 1. The times result from running THERBLIG with $p = 1$ on the same set of profile data: several thousand possibilities were enumerated each run. Line 1 shows that running THERBLIG with $p = 0$ to make an implementation assignment, using that assignment to re-compile THERBLIG, and then running this new THERBLIG over a fixed set of profile data with $p = 1$, resulted in the new THERBLIG taking 34.92 seconds to run (averaged over ten runs). Line 2 shows that creating an implementation assignment for THERBLIG by enumerating all call sites that account for 90% of the runtime resources, resulted in a faster THERBLIG: it required only 32.60 seconds to run. Setting $p = 1$ did not result in a faster program.

Lines 4, 5, and 6 show the result of assigning all the set variables in the program the same implementation. Even though the abstraction is the same (Set), the variables are used differently enough to warrant different implementations. Line 7 is the running time of THERBLIG when every variable is implemented with the default profiling implementations.

The first three lines tell us that a greedy assignment ($p = 0$) yields results comparable to a full enumeration ($p = 1$). This is important because setting $p = 0$ results in a much faster running of the selection algorithm. There were a total of 208 call sites in the THERBLIG sources. When $p = 1$, all 208 were exhaustively enumerated with all possible implementations. When $p = 0$, only thirty, or about 15%, of the call sites were enumerated. In other words, THERBLIG satisfies a 90-15 rule: 15% of its call sites were estimated to account for about 90% of the run time.

We would expect raising p to lower the execution time of the resulting implementation if indeed the evaluation functions correspond to the actual behavior of their corresponding interface functions. Given that $p = 1$ resulted in a slower implementation than $p = 0$ we hypothesize that either the evaluation functions are inaccurate and do not adequately capture the behavior of the implementations, or the difference is in the noise resulting from the fact that we are estimating based on profile data. TYPESETTER does not solve the problem of the reliability of "predictive" test data. In either case, given that the other implementations are worse than THERBLIG's choices — and that the really incorrect implementation *(Set_bmarr)* is five times worse than our 'nearly' correct ones — the anomaly does not appear serious.

Based on the fact that the profiling implementation of Sets is identical to the *Set_slist* implementation with all the profiling code removed, Line 7 (all profiling) and line 4 (*Set_slist*) allow us to conclude that the profiling code slows down the execution of the program about 20%. If the profiling implementation had used bit-mapped arrays instead, then the slowdown would have been worse (line 6). But then, the slowdown would not have been from profiling, but from the unsuitability of the profiling implementation for this particular program.

6 Previous Work

There are two issues: the use of profile data in compilation, and implementation selection. While there are many studies that utilize profile data to analyze experiments or verify analytical techniques, only recently has serious attention been paid to the use of profile data by the compiler. Wall has used profile data in the linking phase

to do register allocation (Wall 1986). Karr explored instruction selection combined with register allocation using profile data (Karr 1984) (see also Morris (Morris 1991). Samples (Samples 1991; Samples 1988), McFarling (McFarling 1989), and Pettis and Hansen (Pettis and Hansen 1990) have looked at using profile data for improving cache performance. Wall has also examined the question of whether real profiles are necessary or whether estimated profiles will do (Wall 1991); his not too suprising conclusion is that real profiles worked "much better" than existing techniques for estimating profile data.

Gilbert Hansen (Hansen 1974) modified a FORTRAN compiler to produce an intermediate representation of a program in one simple, quick pass. When that intermediate form was interpreted, execution counts identified the potential 'hot spots' of the program. An optimizer was invoked only over 'hot spot' code. Each invocation of the optimizer would perform the next level of optimization on the frequently exectuted code. If the code were executed sufficiently often, it would eventually be compiled to machine code. His results showed that the cost of doing a quick, one-pass compilation followed by interpretation and iterative optimization of 'hot spots' was often less than that of doing an equivalent optimization of the original program with a traditional optimizing compiler. Of course, the traditional compiler does not have the benefit of knowing which sections of a program *should* be optimized, so it optimizes the *whole* program. This observation lead directly into our research program. TYPESETTER is the first system to use a general technique for collecting ADT-specific profile data, and using that data to choose implementations.

Low did the original work on implementation selection (Low 1974; Low 1978; Low and Rovner 1976). He used a library of implementations, all written in assembler. His evaluation functions returned the exact number of machine cycles and bytes required on any one invocation of a function. Given that any precision is lost in the estimation of future performance of a real program, TYPESETTER's evaluation functions accept inexactness as inevitable, and assume that programs satisfying the 90-10 rule are skewed enough that such loss of precision will be irrelevant to the final decisions. Furthermore, Low's system did not allow operators to work on multiple representations: a union operator's two operands had to have the same representation. In our approach, a particular implementation function can be assigned to a call site if the actual parameters at the call site can be assigned the types of the implementation function's formal parameters. The Implementor decides which functions are implemented, including mixed-representation functions.

Ramirez (Ramirez 1980) attempted to apply zero-one integer programming to the implementation assignment problem. His solution works only if it assumed that costs of assignments are independent of one another. That is, he assumes that if implementation j is assigned to variable i, then $t(i,j)$, the amount of time required by that assignment, is independent of any other assignments. This is almost never the case, particularly when operators can accept operands with differing implementations (e.g. a *union* of a set implemented as a bitmap with a set implemented as a linked list). TYPESETTER moves the focus from the representation of the variable to the implementation of functions. This allows the interacting costs of assignments to be taken into account.

Sherman's Paragon (Sherman 1985) is an ambitious system that attempts to solve many problems at once, including selection of an implementation of an ADT

based solely on the program text. Profile data could be used, but he does not discuss this in any depth. In the Paragon model, the User (our terminology) is responsible for writing the complete evaluation function (Sherman calls it the *policy* procedure) that selects the implementations of the variables of the program. This puts the onus of selection on the wrong member of the programming team. We have attempted to design a system that puts the onus of implementation evaluation on the Implementor, and selection of implementations for functions and variables on the system.

There has been much research into the synthesis of programs from very high-level descriptions. Rowe's system (Rowe 1976) approached the problem from the direction of selecting an implementation based on an algebraic description of the desired data relations and functionality. In those cases where there did not exist an implementation satisfying the description, Rowe investigated ways of generating an implementation. However, he did not look at the use of profile data in his work. Our use of 'auxiliary declarations' is similar to his declarations of properties of the abstraction.

Barstow's PECOS system (Barstow 1977; Barstow 1985) is a database of rules and deductive heuristics to give a programmer's specification of a program an implementation. Kant (Kant 1981) extended the system to consider rules and heuristics regarding the efficiency of various implementations; she did not investigate the use of profile data.

Kestrel's REFINE system is another example of a high-level approach, and they appear to have paid more attention to the possibility of using profile data (Smith and Goldberg 1986; Smith, Kotik and Westfold 1985).

Selection of implementation based only on static declarations has proven to be difficult and expensive, even when attention is focused on a small set of abstractions, as in the SETL language effort. Work within the SETL project (Dewar et al. 1979; Schwartz et al. 1986) derives implementations from declarations in the language and from analysis; e.g., frequencies are estimated by an analysis of the program text. I know of no work using profile data in the synthesis of SETL programs. The SETL optimizing compiler attempts to determine a good implementation for the set and mapping abstractions in the language (there is only one representation for tuples). The default representation for sets and maps uses hash tables. If the analysis can determine *bases* for the elements of the sets, or if the programmer declares elements to belong to specific bases, then other more efficient implementations are possible for subsets of the bases. A subset can be represented as a bit in the structures for the elements of the bases (if the bit is one, then the element belongs to that subset, if zero, then not). If all elements of a base set are assigned unique integers, then a subset can be implemented as a bit-map. Or a subset might be represented with a separate hash table of pointers into the base set.

Straub's *Taliere* system improves on the optimization phase of the SETL compiler by considering estimates of performance, including symbolic analysis of execution frequencies(Straub 1988). However, since he does not utilize profile data, the User must answer questions[6] of the form *What is the average size of* $s * t$ *in line 215?*; or even *What is the expected number of iterations in an average execution of the loop starting at line 1235?*. Even worse examples of the kinds of dialogue the system

[6] The questions are taken from his dissertation.

forces on the User are questions about probabilities: *What is the probability of the CASE statement of line 1113 taking the alternative of line 1126?* It seems extremely doubtful to me that a User would know this information with any precision or confidence without profile data. And if the profile data exists, then TYPESETTER shows that the compiler can use it directly to answer many, if not most, of these kinds of questions.

Weiss (Weiss 1986) worked on finding types of recursive SETL variables, and presented methods for implementing such structures. However, he is not concerned with selection of 'best' implementations by numeric criteria.

TYPESETTER does not attempt to synthesize programs analytically, nor does it attempt to work with program synthesis at a high level. Rather than seek a Copernican revolution and invent a totally new language in which to specify programs, we sought a more evolutionary approach to give existing languages and systems as much capability as possible.

7 Future Work

We have only scratched the surface with the TYPESETTER prototype. More implementations need to be added to the database of implementations, and more programs need to be written in TYPESETTER to provide further evidence that the greedy assignment heuristic 'scales up'. In addition to issues mentioned throughout the paper, there are other questions that we are pursuing.

While we are convinced that concentrating on information available at call sites yields a simple and effective algorithm, there is useful information that is not associated with a call site but with a specific variable or even with a specific object. For example, we can record at a call site that the elements of a sorted list were added in increasing order, but only for that call site. What may be more relevant is whether there are other call sites that add elements to a specific list, and if *those* elements are added in increasing order also. The current system could keep track of per-variable information with some work, but it loses all per-object information. However, the addition of this extra profile information will be worth while only if the evaluation functions for the ADT interface functions can take advantage of it.

Programmers will always write programs that implicitly use real-world knowledge that need not be expressed in code. TYPESETTER provides a mechanism–optional auxiliary declarations–whereby some of that knowledge can be expressed and used. Some of the optionals currently in use could be deduced by a more integrated implementation of TYPESETTER. For instance, a set of (ASCII) characters can have at most 256 elements. Currently, TYPESETTER has no mechanism for automatically deducing such information.

Currently, TYPESETTER has one profiling implementation for each abstract data type in the library. In the long run, this may be inadequate. Consider the possible implementations of sets as linked lists or as bit arrays. The latter has many possible variation in implementation, each more suitable for certain applications than for others. Choosing which bit array implementation should be used may be decidable only with statistics that are extremely difficult to gather with a profiling implementation using linked lists. In this case, we would almost certainly want another

profiling implementation for bit-mapped sets, one that collects statistics on how sets implemented as bit arrays behave. Then a more accurate choice can be made between the bit array implementations. We need to integrate this hierarchy of profiling implementations into TYPESETTER.

Under the current model, the Implementor is responsible for writing the evaluation functions for an implementation. This is tedious and error prone. It is not yet known how much one badly written evaluation function can affect the final implementation of a program. More analysis is needed into how imprecise evaluation estimates can be and still be useful.

References

Barstow, D. (November 1977): Automatic Construction of Algorithms and Data Structures using a Knowledge Base of Programming Rules. PhD Dissertation, Stanford University

Barstow, D. (January 1985): On Convergence Toward a Database of Program Transformations. ACM Transactions on Programming Languages and Systems **7**, 1–9

Dewar, R.B.K., Grand, A., Liu, S-C., Schwartz, J.T. (January 1979): Programming by Refinement, as Exemplified by the SETL Representation Sublanguage. ACM Transactions on Programming Languages and Systems **1**, 27–49

Ellis, M., Stroustrup, B. (1990): The Annotated C++ Reference Manual. Addison Wesley, Reading, MA

Gilbreth, Jr., F.B., Carey, E.G. (1948): Cheaper by the Dozen. T.Y. Crowell, New York

Hansen, G.J. (1974): Adaptive Systems for the Dynamic Run-time Optimization of Programs. PhD Dissertation, Carnegie-Mellon University, Pittsburgh, PA

Kant, E. (1981): Efficiency Considerations in Program Synthesis. Stanford University, PhD Dissertation

Karr, M. (June 1984): Code Generation by Coagulation. Proceedings of the ACM-SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices **19**

Knuth, D.E. (1971): An Empirical Study of FORTRAN Programs. Software–Practice Experience **1**, 105–133

Knuth, D.E., Stevenson, F.R. (1973): Optimal measurement points for program frequency counts. BIT **13**, 313–322

Low, J.R. (August 1974): Automatic Coding: Choice of Data Structures. Computer Science Department, Stanford University, PhD Dissertation, Technical Report CS-452

Low, J.R. (May 1978): Automatic Data Structure Selection: An Example and Overview. Communications of the ACM **21**, 376–385

Low, J.R., Rovner, P. (January 1976): Techniques for the Automatic Selection of Data Structures. Conference Record of the Third ACM Symposium on Principles of Programming Languages

McFarling, S. (April 3-6, 1989): Program Optimization for Instruction Caches. Symposium on Architectural Support for Programming Languages and Operating Systems, Boston, MA

Morris, W.G. (June 26-28, 1991): CCG: A Prototype Coagulating Code Generator. Proceedings of the ACM-SIGPLAN 1991 Conference on Programming Language Design and Implementation **26**, 45–58

Pettis, K., Hansen, R.C. (June 20-22, 1990): Profile Guided Code Positioning. Proceedings of the ACM-SIGPLAN 1990 Conference on Programming Language Design and Implementation **25**, 16–27

Ramirez, R.J. (March 1980): Efficient Algorithms for Selecting Efficient Data Storage Structures. Faculty of Mathematics, University of Waterloo, PhD Dissertation, Technical Report CS-80-18

Rowe, L.A. (1976): A Formilization for Modelling Structures and the Generation of Efficient Implementation Structures. University of California, PhD Dissertation

Samples, A.D. (April 1991): Profile-driven compilation. Computer Science Division, EECS, University of California, Berkeley, Technical Report UCB/CSD 91/627

Samples, A.D. (October 1988): Code Reorganization for Instruction Caches. Computer Science Division, EECS, University of California, Berkeley, Technical Report UCB/CSD 88/447

Samples, A.D., Hilfinger, P. (October 1990): Profile-Driven Compilation. InfoCon '90, Tokyo, Japan. [The pages are out of order in the proceedings; the order should be: 169, 172,171,170,174,173,175,176.]

Schwartz, J.T., Dewar, R.B.K., Dubinsky, E., Schonberg, E. (1986): Programming with Sets: An Introduction to SETL. Springer Verlag, Berlin

Sherman, M.S. (1985): Paragon: A Language Using Type Hierarchies for the Specification, Implementation and Selection of Abstract Data Types. (Lecture Notes in Computer Science, vol. 189) Springer Verlag, Berlin

Smith, D.R., Goldberg, A. (November 1986): Towards a Performance Estimation Assistant. Palo Alto, CA. KES.U.86.10

Smith, D.R., Kotik, G.B., Westfold, S.J. (November 1985): Research on Knowledge-Based Software Environments at Kestrel Institute. IEEE Transactions on Software Engineering **SE-11**

Straub, R.M. (May 1988): Taliere: An Interactive System for Data Structuring SETL Programs. PhD Dissertation, Courant Institute of Mathematical Sciences, NYU

Wall, D. (June 1986): Global Register Allocation at Link Time. Proceedings of the ACM-SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN Notices **21**, 264–275

Wall, D.W. (June 26-28, 1991): Predicting Program Behavior Using Real or Estimated Profiles. Proceedings of the ACM-SIGPLAN 1991 Conference on Programming Language Design and Implementation **26**, 59–70

Weiss, G. (March 1986): Recursive Data Types in SETL: Automatic Determination, Data Language Description, and Efficient Implementation. Courant Institute of Mathematical Sciences, NYU, Department of Computer Science, Technical Report 102