Type Transformation and Data Structure Choice

J. Cai NYU
P. Facon CNAM, Paris
F. Henglein NYU
R. Paige NYU
E. Schonberg NYU

January 12, 1994

Categories and Subject Descriptors: D.1.2[Programming Techniques]: Automatic Programming; D.3.2[Programming Languages]: Language Classifications - very high level languages; D.3.3 Language Constructs - Abstract Data Types; D.3.4 Processors - Optimization; F.3.2 [Logics and Meanings of Programs]: F.3.1 Specifying and Verifying and Reasoning about Programs; Semantics of Programming Languages - Operational Semantics; F.3.3 Studies of Program Constructs - Type Structure; E.2 [Data Storage Representations]

Type Transformation and Data Structure Choice

Abstract

Program analysis for typings, for set inclusion and membership relationships, and for universal sets to support data structure selection was a major goal of the SETL project and involved perhaps 50 person-years of design and implementation. Because that approach was so comprehensive, its data structure algorithms used heuristics leading only to expected improvements in performance. Despite serious pragmatic intentions, the optimizer that was eventually implemented was too large - essentially 20,000 lines of SETL source code - and was never fully operational. We believe that the results of that project fell short of expectations, because the necessary program analysis of a weakly typed, unstructured, and potentially low-level language such as SETL required inferences that were too difficult to obtain in any practical way.

This paper describes a top-down approach to data structure selection in which typings and universal sets are easily inferred in a high level, declaration free, strongly typed, functional problem specification language. All remaining analysis to facilitate data structure choice is obtained by a simple deductive process carried out by program transformations that compile these high level specifications into efficient RAM code. A specific transformation to Ada is presented and illustrated using the example of database attribute closure. Our approach is somewhat more restricted than that of SETL, but it rests on formal foundations with data structure selection algorithms that simulate a set machine on a RAM in real-time.

1 Introduction

In his Turing Award address Tarjan said,

> "Conventional programming languages force the specification of too much irrelevant detail, whereas newer very-high-level languages pose a challenging implementation task that requires much more work on data structures, algorithmic methods, and their selection."[Tar87]

In his Turing Award interview he continued,

> "It would be wonderful in the long run to have some kind of supercompiler that would select, off-the-shelf, the appropriate data structure to plug in to implement very high-level quasialgorithmic specifications. Ultimately, things have to go in this direction."[Fre87]

In order to increase productivity of the most difficult kinds of software, such as a high-performance optimizing compiler or a library of computational geometry code, one must

1. automate major aspects of algorithm design;
2. automate the translation of algorithm specification into reasonably efficient code;
3. ensure that the complex software ultimately designed is correct.

This paper describes some progress in the three preceding areas based on a synthesis of ideas from type theory, algorithm design, and software engineering. We show a novel way in which typings inferred in a high level, declaration free, strongly typed, functional specification language can be transformed into lower level subtypings that provide concrete data structures for an implementation language (e.g., one that is explicitly typed like Ada). The low level code produced by our method is guaranteed to have (1) all variables initialized before they are used, (2) all array accesses in bounds, (3) no dangling references, (4) no dereferences of nil pointers, and (5) no type errors occurring at runtime.

Our type transformation rests on the discovery of finite universal sets, called *bases*, to be used for avoiding data replication and for creating aggregate data structures that implement logical associative access operations using simpler cursor or pointer access. *Principal* (i.e., unique best) types and bases are inferred from the high level specification and are extended to principal types and subtypes for an intermediate language implementation P in the presence of an abstraction of P called the *object flow graph* (OFG). Data structures that are in some sense optimal (but not unique) are inferred from the OFG and from the subtypings provided from previous analysis.

Our method of choosing efficient representations for sets, maps, and other structured datatypes embodies a new algorithm design technique based on the real-time simulation of an abstract sequential set machine on a uniform cost sequential RAM[AHU74]. We believe that further development of this approach could lead to the more ambitious data structure compiler envisioned by Tarjan.

Section 2 describes the set machine language SML and data structure design principles that support the real-time simulation of SML on a RAM. In Section 3 we describe the specification language $SQ2^+$, its type model, and the method of type inference. Discovery of bases and subtype inference is found in Section 4. Transformation from $SQ2^+$ to SML, transformational analysis of the OFG, and extending types and subtypes to new variables in the SML is the topic of Section 5. Section 6 is about data structure inference. Section 7 investigates further data structure refinements into efficient Ada storage structures. Section 8 reports some empirical results from an implementation. Section 9 mentions some useful generalizations and future work.

2 Real Time Simulation of a Set Machine on a RAM

In this section we describe the data structure design method on which our program optimization technique is based. This method by itself forms the basis for a theory of naive data representation and could greatly simplify the presentation found in the first seven chapters of Aho, Hopcroft, and Ullman [AHU83]. However, the main goal in this paper is to describe how to incorporate data structure selection as part of a high level programming language optimizer.

We first define a simple set machine language SML and its associated abstract model of complexity. For each SML primitive operation q, we define a worst case asymptotic time bound $O(f_q)$ for performing q on an idealized set machine. Next, we define a transformation T that uses coarse data structures to simulate SML programs (satisfying sufficient conditions, defined more precisely later) on a uniform cost sequential RAM in real time. Transformation T turns each primitive operation q in the SML program into a functionally equivalent sequence of RAM operations whose worst case asymptotic time (expected time when the sufficient conditions are not met) and space is the same as the set machine complexity. Consequently, if an SML program P has an $O(g(n))$ space and $O(h(n))$ time asymptotic worst-case complexity on the set machine, then $T(P)$ has the same asymptotic worst-case time (expected time when the sufficient conditions are not met) and space bounds on a RAM.

It is important to note that our simulation depends strongly on the presentation of the input data and that the cost of preconditioning the input to facilitate RAM computation is not included in our RAM complexity. For example, in the style of standard algorithm texts (see [AHU74]), we could assume that the input data forms a set without repetition of elements or represents a graph as an adjacency list. However, we disallow sorting and other operations that order data as a form of preconditioning.

Our set machine language SML includes conventional unit space datatypes such as integer and boolean, fixed length heterogeneous tuples (i.e., records with fields identified by numerals), homogeneous dynamic sequences, and finite homogeneous dynamic sets, where a set of ordered pairs is regarded as a multivalued-mapping. Although our data structure representations can be used to implement tuples and sequences, we can, without loss of generality, restrict our attention to sets and maps.

It is useful to divide up the primitive SML operations into the following four categories:

1. *Retrieval* operations select an arbitrary value from a set.
2. *Initialization* operations assign a set to be empty.
3. *Addition* operations add a new element to a set.
4. *Associative* access operations locate a given value within a set.

See Table 1 for a list of primitive operations and their defined complexities grouped according to the categories mentioned above. SML also contains conventional unit time boolean and arithmetic operations, and a full repertoire of control statements that include while-loops, for-loops, conditionals, and goto's. We assume that assignment statements are destructive to the original left-hand-side value. We also combine copy/value and pointer semantics as follows. Assigning a boolean or arithmetic value v makes a new copy of v, but assigning an aggregate value v makes v shared. For example, if T and S are sets, then the assignment T **with** $:= S$ results in the value of S being shared by the variable S and by an element of T. A subsequent assignment to S would then modify T as a side effect. In fact, if we retrieve an arbitrary element Q of T, then modifying Q could change both T and S.

These assignment semantics are easy to implement efficiently, but they make SML programs difficult to understand. However, we will show that SML programs generated by certain transformations from high level SQ2+ specifications can always be interpreted equivalently in terms of the more comprehensible copy/value semantics.

Real-time simulation of SML on a uniform cost sequential RAM cannot always succeed, because arbitrary membership tests $x \in S$ for dynamic sets S stored in linear space require $\Omega(\#S)$ comparisons in the worst case [Knu72]. Such failure could be overcome by rewriting membership tests as explicit searches (e.g., linear search) that can be simulated in real-time on a RAM. However, this approach is likely to increase the SML time complexity. Another approach, which is similar to the default implementation in SETL, is to store each set S as a hash table and be satisfied with an expected time simulation. We prefer to fall back on hashing only after real-time simulation fails.

Four basic kinds of data structures are discussed. The simplest one for implementing sets is a doubly-linked list with pointers to the first and last list cell. Each list cell stores an element of the set. This representation, whose elements are said to be *unbased*, supports the real-time simulation of retrieval, addition, and initialization (re-initialization can also be handled by amortizing garbage collection), but, in general, not access.

The problem with associative access can be illustrated with the following simple example:

```
while (∃ x ∈ S)
    ...
    Q less:= x
    ...
end
```

Operation	Definition	Complexity
	Retrievals	
$\ni x$	arbitrary choice	$O(1)$
$\exists x \in s$	boolean valued existential quantifier with side effect	$O(1)$
$f(y)$	x, if $f\{y\} = \{x\}$, undefined otherwise	$O(1)$
(**for** $x \in s$)	iterate for each element x in s	$O(\#s)$
...		
end		
	Initialization	
$f := \{\}$	assign empty map	$O(1)$
$s := \{\}$	assign empty set	$O(1)$
	Addition	
s **with** $:= x$	set element addition	$O(1)$
	Access	
$x \in s$	set membership test	$O(1)$
$f(x)$	y, if $f\{x\} = \{y\}$, undefined otherwise	$O(1)$
$f\{x\}$	$\{y : [u, y] \in f \mid u = x\}$	$O(1)$
$f(x) :=$	indexed assignment to function	$O(1)$
$f\{x\} :=$	indexed assignment to map	$O(1)$
s **less** $:= x$	set element deletion	$O(1)$

Table 1: SML Primitives

In the preceding code, if S and Q are both implemented as doubly linked lists, then retrieving an arbitrary value from S and storing it in x can be done in O(1) time. However, the subsequent search needed to locate this value within Q in order to delete it cannot, in general, be achieved in unit time. Only when sets S and Q are the same identifiers, can we always ensure that the associative access (which in this case is called a $self - access$) can be executed in unit time.

In order to solve the associative access problem mentioned above, we follow the approach found in [SSS81] and [PH87], where values common to both sets S and Q are stored in one place - in a finite universal set called a *base*. Consequently the unit time retrieval from S locates the value within Q as well. More generally, we use a finite universal set B as the base for S and Q and maintain the invariant

$$S \cup Q \subset B$$

To maintain this invariant we represent B and Q as a set of records, each record containing a B and a Q field. The elements of B are stored in the B field and serve as the key. Given any record whose B field has the value x, the Q field in this record stores the undefined value Ω if x does not belong to Q. Those records whose B field values belong to Q are connected by a doubly linked list stored within their Q fields. There are also $first$ and $last$ pointers to the first and last records of the Q field list. Set S is represented as a doubly linked list of pointers to records whose B fields store the elements of S. We also use $first$ and $last$ pointers for the S list.

Objects aggregated around the same base are said to be *compatible*. Hence, the elements of S and Q are compatible. We say that the elements of S are *weakly based* and the elements of Q are *strongly based* on B (see Figure 1).

Weakly and strongly based representations support real-time simulation for our four basic forms of primitive operations. As in the case of unbased sets, retrievals from sets whose elements are either weakly or strongly based can be performed in unit time. Also, when an object x is compatible with the elements of a set S, then x can be added to S in unit time. When the elements of S are also strongly based, then x can be used as a search argument to perform a unit time associative access on S. However, for the same reason as when S is unbased, when the elements of S are weakly based, then we can only perform self-access operations in unit time. Initialization (and also re-initialization) for sets whose elements are weakly based is similar to the unbased case. For sets whose elements are strongly based, we can either amortize the linear time initialization costs, or obtain unit time initialization (and re-initialization) by using the solution to exercise 2.12 of Aho, Hopcroft, and Ullman's book [AHU74].

Any object storing a value belonging to a base B can be weakly based on B. Thus, the elements of any set that is a subset of a base B can be weakly based on B. To avoid complications we need to be somewhat more restrictive with strongly based representations. Any set that is a member of another set is said to be *nonsimple*. Because the range

of a map is organized as a collection of image sets, image sets are examples of nonsimple sets. Because the number of nonsimple sets is data dependent in general, we stipulate that their elements can only be weakly based but not strongly based. If we allowed them to be strongly based and these sets were sparse, then an asymptotic increase in space could arise. Thus, for example, our method can choose space efficient data structures such as adjacency lists but not adjacency matrices.

It is useful to illustrate the preceding ideas with the simple example of graph reachability. This problem inputs a set of edges e, represented as a set of ordered pairs, and a subset w of vertices. The problem is to find the set of vertices r reachable along paths in e from w. The SML code just below runs in worst case time O(#e) on a set machine.

$t3 := \{\}$
(**for** $x \in w$)
    $t3$ **with**:= x
end
$r := \{\}$
(**while** $\exists a \in t3$)
    (**for** $y \in e\{a\}$)
        if $y \notin r$ **and** $y \notin t3$ **then**
            $t3$ **with**:= y
        end
    end
    $t3$ **less**:= a
    r **with**:= a
end

Clearly, all assignments in the preceding code can be interpreted with copy/value semantics. Consequently, every assignment to a set can be performed destructively, and no costly hidden copy operation due to shared data needs to be performed. If we let $v = w \cup$ **domain** $e \cup$ **range** e be our base, then it is easy to prove the program invariants $t3, r \subseteq v$ and $a, x, y \in v$. Consequently, the SML code can be simulated in real-time if the elements of **domain** e, r, and $t3$ are strongly based on v (to handle the three associative access operations), and a, x, y, and the elements of **range** e are weakly based on v (to satisfy base compatibility constraints).

Note that universal sets were important in the preceding example by eliminating replicated values and shortening access paths. This is, of course, a well-known major efficiency seeking goal in dynamic databases[Dat82,Wie83,Ull80], where modifying a database in which distinct values may be stored in many files often results in forced redundant modifications to each of these data files.

In the following pages we will show how analysis for universal sets, inclusion and membership tests, and data structure choice can be achieved in a straightforward and completely automatic way.

3 Specification language and type inference

The problem of automatic data structure selection and aggregation for a set theoretic language has been studied before for SETL[Sch75b,SSS81,DGC$^+$79,FSS83]. However, that work took a heuristic approach that aimed for reasonable expected performance rather than a real-time simulation. Also, their theoretical results (e.g., the notion of value flow[Sch75a]), which we draw on, were far more impressive than their application within the SETL optimizer. We believe that data structure inference in an unstructured language such as SETL with complex control and data flow is essentially intractable.

Our alternative top down approach finesses the problems faced in SETL optimization by doing an initial analysis of problem specifications in a very high level functional language in which crucial semantic information is localized and easily obtainable from the syntax. Through a largely top down process, we refine and augment this initial information while transforming the specification until we produce a program in which data structures can be determined with a minimal amount of effort.

Our problem specification language *SQ2*$^+$ is functional, strongly typed, Turing complete, and declaration free. It contains conventional boolean and integer datatypes augmented with a full repertoire of finite set theoretic operations plus least and greatest fixed point operations, which are defined in terms of operational semantics.

Because this paper is primarily concerned with data structure selection, we use a simple strong typing model with no union types, no overloading, and no user defined parametric polymorphism. The (first-order) type expressions are all the expressions generated by the following grammar:

```
T       ::=     int  |  Bool   |
                t1, t2, ...    |       type variables
                [T, ... , T]   |       records
```

Expression	Definition
$s \cup t$	set union, intersection, difference
$\cap$	
$-$	
$s \subseteq t$	subset test
$s = t$	equality test
$\#s$	set cardinality, record size, sequence size
$\ni s$	arbitrary choice
$x \in s$	membership test
$\notin$	
$\{\}$	empty set
$\{x_1, ..., x_n\}$	enumerated set
$\{e(x) : x \in s \mid k(x)\}$	set former
$\forall s \in s \mid k(x)$	boolean valued quantifier
$\exists$	
$f(x)$	y, if $f\{x\} = \{y\}$, undefined otherwise
$f\{x\}$	$\{y : [u, y] \in f \mid u = x\}$
$f[x]$	$\bigcup_{x \in s} f\{x\}$
domain f	$\{x : [x, y] \in f\}$
range f	$\{y : [x, y] \in f\}$
$\mathbf{LFP}_0\ x.e$	least fixed point
$\mathbf{GFP_1} x.e$	greatest fixed point

Table 2: **SQ2$^+$ set expressions**

```
<T>          |     sequences
{ T }              finite sets
```

Type variables are used for "input" polymorphism and built-in polymorphic operators such as cardinality (#). To avoid overloading we will assume that the cardinality operator is subscripted by the coarse type of its argument (e.g., set or sequence). A map is defined to be a set of pairs; that is, it is of type $\{[T1, T2]\}$ where $T1$ is the element type of the domain and $T2$ is the element type of the range.

A type expression t represents a set of values $\mathbf{set}(t)$. We denote the fact that *SQ2*$^+$ expression e has type t using the notation $e : t$ (which means that $e \in \mathbf{set}(t)$ for any input assignment). We denote the fact that $t1$ is a subtype of $t2$ by writing $t1 \leq t2$ (which implies that $\mathbf{set}(t1) \subseteq \mathbf{set}(t2)$). Subtypes are described later.

Table 2 lists the expressions of **SQ2**$^+$. We provide operational semantics for **SQ2**$^+$ by using SML implementations. The semantics of fixed point expressions $\mathbf{LFP}_B s.e$ (i.e. least fixed point of expression e with respect to parameter s that is greater or equal to B) is somewhat tricky. These expressions are only defined for partially ordered datatypes int, Bool (where false $\leq$ true), and sets (under set containment). As in [CP89] we define $\mathbf{LFP}_B s.e$ in terms of a syntactically defined decidable subclass of monotone (i.e., $x \leq y$ implies that $f(x) \leq f(y)$), inflationary (i.e., $x \leq f(x)$), and computable expressions. (The class of monotone *SQ2*$^+$ expressions is undecidable[Gur84].). If expression e is set valued, then the semantics of $\mathbf{LFP}_B s.e$ is given by the meaning of the following SML code (which is based on [CC79,CP89]):

$S := B$
(**while** $\exists x \in e(S) - S$)
    S **with** $:= x$
end

SQ2$^+$ is a strongly typed variant of a weakly typed specification language *SQ1*$^+$ used to study fixed point computation as a transformational programming paradigm[PH87,CP89]. Just as *SQ1*$^+$ was proved to be Turing complete [CP89] by simulation, we have

Theorem 1 SQ2$^+$ *is Turing-complete.*

A type system for *SQ2*$^+$ in the spirit of the Simply Typed Lambda Calculus is presented in Appendix B. It is a specialization of the "patterned" form of presentation from [Lei83] and [FM88]. As a consequence of the results by Hindley [Hin69] and Curry [Cur69] (who solved type inference by solving term equations with first order unification) this

calculus has a principal typing property, which states that every well-typed term admits a most general assignment of types to program variables.

To illustrate how type inference can be modelled by solving equations, consider expression $S \cup T$. Our type calculus would generate the following type equations, solvable by unification:

$S : t1$
$T : t2$
$t1 = \{t3\}$
$t2 = \{t4\}$
$t1 = t2$

To illustrate type error detection, we let $pow(x)$ represent the power set of x. Then the expression **LFP** $x.pow(x)$ leads to the equation $\{t\} = \{\{t\}\}$, which reduces to $t = \{t\}$, which triggers an occurs check and, hence, a type error.

Theorem 2 *Type Inference for* SQ2$^+$ *has the Principal Typing property and is solvable in Linear Time*

Proof See Hindley and Curry for reducing type inference to unification, in which the most general unifier yields the most general typing. The complexity follows from the fact that the size of the term equations generated by *SQ2*$^+$ type inference is directly proportional to the length of the expression being typed, and from the linear time unification algorithm due to Patterson and Wegman[PW78]. □

One motivation for our approach is the belief that strong typing is both most effective and most useful for languages at the highest level of abstraction. It is most effective, because

1. Clarity is a common goal of high level specification and strong typing. The nonprocedural, functional style of abstract specification both facilitates the kind of fact gathering required of type inference, and at the same time is least crimped by restrictions imposed by a strong typing discipline. In contrast, the emphasis on efficiency considerations in low level programming foster complex control flow, which confounds strong typing disciplines.

2. The powerful and diverse datatypes in abstract specification make important semantic information localized in the syntax, and hence, recoverable by mechanical inference. However, in low level programming deep semantic information is dispersed and difficult to recover.

Strong typing is most useful, because

1. Strong typing provides a kind of quality control that is vital to specification-level programming, where meaning is highly sensitive to the slightest syntactic change. The global meaning of low level code is much less sensitive to local error.

2. High level datatypes make the kind of consistency between operators and their arguments ensured by strong typing semantically more meaningful. However, the simplicity and uniformity of datatypes in the lowest level programming make strong typing less meaningful and makes it harder to detect errors.

We will illustrate our techniques using the problem of attribute closure in relational databases (informally discussed in [PH87]). For this problem input consists of a finite set X of attributes and a set f of functional dependencies represented as a multi-valued mapping from a set of sets of attributes to a set of attributes; that is, f maps each set of attributes Y in its domain into a set $f\{Y\}$ that functionally depends on Y. The attribute closure is the smallest set S that includes X and also includes $f\{Y\}$ whenever it includes Y. The following is a formal *SQ2*$^+$ specification of the attribute closure problem.

(1) $\quad X^+ \equiv \text{LFP}S.X \cup f[\{Y \in \textbf{domain}f \mid Y - S = \emptyset\}]$

Type inference will produce the following (principal) typing:

$X : \{t\}$
$f : \{[\{t\}, t]\}$
$X^+ : \{t\}$

where t is a type variable that is treated like a generic type, and $\{t\}$ denotes the type of finite sets with element type t.

4 Type transformation and the discovery of universal sets

As our example of the preceding section illustrates, *SQ2*$^+$ types are abstract data structures with a hierarchical organization of accessible units that store retrievable data values. We call these units *objects* and denote them by typed *SQ2*$^+$ expressions $e : t$, which can be used to represent sets $\mu(e)$ defined according to the following rules:

1. If $e : t$ is a typed *SQ2*$^+$ expression, then $e : t$ is an object representing the set $\mu(e) = \{e\}$.

2. If $e : \{t\}$ is an object and $t \neq [t1, t2]$, then so is $\ni e : t$, and $\mu(\ni e) = \bigcup_{x \in \mu(e)} x$.

3. If $e : \{[t1, t2]\}$ is an object, then so are $\mathbf{domain}e : \{t1\}$ and $\ni \mathbf{range}e : t2$; $\mu(\mathbf{domain}e) = \{\mathbf{domain}x : x \in \mu(e)\}$ and $\mu(\mathbf{range}e) = \{\mathbf{range}x : x \in \mu(e)\}$.

4. If $e : [t1, ..., tk]$ is an object, then so is $ei : ti, i = 1, ..., k$, and $\mu(ei) = \{xi : x \in \mu(e)\}$.

5. If $e :< t >$ is an object, then so is $\ni e : t$, and $\mu(\ni e) = \bigcup_{x \in \mu(e)} \{y : y \in \mu(e)\}$.

It is useful to define the set of *input* objects as the smallest set containing:

1. objects $v : t$, where v is an input variable or constant;

2. any object derivable from (1) using the preceding rules.

The input objects for our example are: (1) $X : \{t\}$, which denotes the set $\{X\}$; (2) $\ni X : t$, which denotes X; (3) $f : \{[\{t\}, t]\}$, which denotes the set $\{f\}$; (4) $\mathbf{domain}\ f : \{\{t\}\}$, which represents $\{\mathbf{domain}f\}$; (5) $\ni \mathbf{range}\ f : t$ which represents the range of f; (6) $\ni \mathbf{domain}f : \{t\}$, which denotes the domain of f; and (7) $\ni\ni \mathbf{domain}f : t$, which represents the union of the elements of the domain of f.

Let us now augment our initial types with the following new types defined in a lattice:

1. corresponding to every subset $\{e1 : t, e2 : t, ..., ek : t\}$ of input objects of identical type t is a new distinct type, called a *base* type b such that $\mathbf{set}(b) = \bigcup_{i=1,..,k} \mu(ei)$ and $b \preceq t$; if $b1$ and $b2$ are two base types, then $b1 \preceq b2$ iff the subset of input objects associated with $b1$ is contained in the subset associated with $b2$;

2. new base types are created by set, sequence, and tuple constructors with the ordering $b1 \preceq b2$ iff $\{b1\} \preceq \{b2\}$ iff $< b1 > \preceq < b2 >$, and $bi \preceq ci, i = 1, .., k$ iff $[b1, ..., bk] \preceq [c1, ..., ck, ...]$;

Before we go on to describe how $SQ2^+$ expressions can have base types, it is useful to distinguish *created* objects from *input* objects. A created object stores new values generated by operations and not copied from input objects. For example, x^3 is created as is $S \cup T$, pow(S), and $\ni$ pow(S). However, if $\ni S$ and $\ni T$ are input objects, then so are $\ni (S \cup T)$ and $\ni\ni$ pow(S).

Analysis for universal sets, or bases, makes use of the results of type inference and input object determination. For simplicity we restrict a base to be the union of values from input objects of the same type. Base analysis is a partitioning of the input objects using an approach, which, like type inference, reduces to a linear time unification problem yielding principal bases.

The term equations that are generated in base analysis reflect more particular information than types about possible overlapping of values stored within objects. For example, if X and Y are two sets whose elements are input objects, we will assume that these sets overlap (hence, have the same base) if we see set operations such as $X \cup Y$ or $X - Y$. If the elements of set X and the elements of **domain** f are input objects, we will assume that these sets overlap (and so have the same base) if we see the image set operation $f[X]$. Because of interactions between X and **domain** f, the image set expression also indicates that **domain** f must be a subset of some base. Finally, when E is a set expression with elements that are input objects, then **LFP** $X.E$ tells us that the elements of X are input objects with the same base as the elements of E.

If we apply rules such as these to the Attribute Closure example, we can conclude that the elements of X and **range** f together with the elements in all the sets belonging to **domain** f must belong to the same base. Hence, the union of all these sets form a single base. Also, **domain** f forms a base by itself. That is,

$A = X \cup \mathbf{range}f \cup (\bigcup_{Y \in \mathbf{domain}f} Y) : \{t\}$

$I = \mathbf{domain}f : \{\{t\}\}$

These bases, being set valued, can be regarded as types and used to refine the initial typings to form based subtypings that reflect the new information. Thus, we obtain,

$A : \{t\}$

$I : \{\{A\}\}$

$X : \{A\}$

$f : \{[I, A]\}$

$s : \{A\}$

Note that the following subtype relations hold:

$A \leq t$ and $I \leq \{A\} \leq \{t\}$

The problem of inferring bases can be succinctly formalized in a subtype calculus, adapted from [Mit83] and [FM88], that is systematically derived from the original type system by adding type coercions to typing statements and axiomatizing coercion as type containment. Consequently, we have

Theorem 3 *Base Subtype Inference for SQ2+ has the Principal Typing property and is solvable in Linear Time*

Proof Similar to the proof for Type Inference.

5 Transformation From $SQ2^+$ to SML

SML serves as an intermediate language that facilitates code and type transformation from $SQ2^+$ to efficient RAM code (which can be represented by a variety of conventional lower level languages). In a later section, we will illustrate our method using Ada as the target language. SML is also used to provide an operational semantics for $SQ2^+$ and to help define a class of acceptable $SQ2^+$ to SML transformation, which is the topic of this section.

As previously described, SML has the same datatypes as $SQ2^+$, but is limited to lower level set operations. Recall that these set operations can be divided into four categories - retrieval, addition, initialization, and associative access.

Let us continue with the example of attribute closure and illustrate a single naive transformation from $SQ2^+$ to SML based purely on the operational semantics of $SQ2^+$. Because the function subpart of specification (1) is monotone, the fixed point exists and can be computed by the iterative procedure (corresponding the the operational semantics of **LFP**) just below:

(2) $S := \{\}$
($\mathbf{while} \exists z \in (X \cup f[\{Y \in \mathbf{domain} f \mid Y - S = \{\}\}]) - S)$
S **with** $:= z$
end

Code (2) can easily be implemented in SML by straightforward bottom up evaluation of its subexpressions,

(3) $T1 = Y - S$
$T2 = \{Y \in \mathbf{domain} f \mid T1 = \{\}\}$
$T3 = f[T2]$
$T4 = X \cup T3$
$T5 = T4 - S$

according to their operational semantics. The resulting code appears in Appendix A. This code preserves the meaning of expression (1) by definition of the operational semantics of $SQ2^+$. Since our operational semantics is crafted so that every SML definition for an $SQ2^+$ expression can be interpreted equivalently under copy/value semantics for assignments, the code in Appendix A has this property too. This is critical to our real time simulation, which requires that sets be assigned destructively, and hidden copy operations due to shared data can be avoided.

The naive transformation based on operational semantics also allows us to extend the base subtypes for the input variables to subtypes for the variables in the code it produces. A simple subtype analysis of the subexpressions T1,. . . ,T5 tells us

$T1 : \{A\}$
$T2 : \{I\}$
$T3 : \{A\}$
$T4 : \{A\}$
$T5 : \{A\}$

Base subtype information for the remaining variables of the SML code can be determined from the way in which they are related to these main variables already typed. Clearly, the code produced is well typed.

However, in order for the naive transformation to be suitable for data structure selection, the subtypes determined for the code it produces must satisfy one additional property. We consider an 'object flow graph' that abstracts potential interactions between objects in an SML program at runtime. For each retrieval operation in an SML program, where an object x is retrieved ¿from a set s, we draw an edge $s\exists \rightarrow x$; for each SML operation where x is added to set s, we draw an edge $x+ \rightarrow s$; if x is used to perform an associative access on s, then we draw $x \in\!\!\rightarrow s$. The object flow graph is related to but simpler than Schwartz's value flow analysis [Sch75a].

Our naive transformation can generate an object flow graph recursively based on fragments of such a graph associated with the operational semantics of an $SQ2^+$ expression. See Figure 2 for the object flow graph associated with the code in Appendix A.

We say that base subtype analysis for an $SQ2^+$ to SML transformation is *valid* relative to the object flow graph if the OFG satisfies the following conditions:

- If $s : \{B\}$ and $x+ \rightarrow s$, then $x : B$.
- If $x : B$ and $s\exists \rightarrow x$, then $s : \{B\}$.
- For each edge $x \in\!\!\rightarrow s$, $x+ \rightarrow s$, or $s\exists \rightarrow x$ in which x and the elements of s are input objects, then x and the elements of s must belong to the same base.

The preceding validity conditions identify a unique optimal and sound partition (as a least fixed point) of the input objects into bases, and gives rise to a lattice of partitions, where the coarsest such partition is the one in which each base is the union of all input objects with identical type.

Since determining object flow graphs and detecting input objects at compile time is undecidable, our subtype transformations must be semantically incomplete. However, with respect to a conservative approximation of object flow graphs

(where more edges could be detected) and input objects (where fewer such objects would be detected), relative validity is a reasonable condition to impose on any $SQ2^+$ to SML transformation.

We say that an $SQ2^+$ to SML transformation is *acceptable* if it is meaning preserving, produces code that is well typed, produces code with copy/value semantics, and produces a valid OFG. Based on the preceding discussion, we have,

Theorem 4 *The naive transformation is acceptable.*

The naive transformation has the disadvantage of producing code with repeated calculations of potentially costly expressions associated with variables T1,...,T5. We can sometimes overcome this problem by using less expensive incremental calculations to keep the values of these expressions stored within variables T1, ...,T5. (An informal discussion of finite differencing applied to the attribute closure example is found in [PH87].) This approach has been formalized by Paige and Koenig [PK82] as a general meaning preserving finite differencing transformation. We can also show that it produces code that is well typed, has copy/value semantics, and produces a valid OFG. Thus, we have,

Corollary 5 *Transformation by finite differencing is acceptable.*

6 Data structure inference

In this section we show how base subtypes obtained from previous analysis can be further refined into data structure subtypes that indicate storage representations suitable for a low level implementation language such as Ada. These data structures have two main goals:

1. to reduce space by reducing replicated values and aggregating data;
2. to support O(1) time associative access (with worst case time as a primary goal and expected time as a secondary goal) without sacrificing space.

The type of an object that is weakly based or strongly based on a base B is denoted by $B-w$ or $B-s$ respectively. Under certain conditions real-time simulation gives us only O(1) expected time instead of worst case time on a RAM. When this is the case the expected time results from hashed data structures. Weakly based, strongly based, and unbased sets can be hashed, which would be denoted by annotating the set element type with the symbol h; e.g., if B is a base, then $\{B-wh\}$ represents a set whose elements are weakly based on B and is stored in a hash table.

Data structure inference for OFG's produced by acceptable transformations is based on the following three rules. If rule 1 does not apply, then real-time simulation succeeds; otherwise, simulation only achieves expected time.

1. If $x \in\!\!\rightarrow s$ and either x or the elements of s are created objects, or if s is nonsimple, then set s is represented by a hash table.

2. If $x \in\!\!\rightarrow s$ and $s:\{B\}$, then $x:B-w$ and $s:\{B-s\}$ if s is simple and $s:\{B-w\}$ otherwise.

3. If $x+ \rightarrow s$ or $s\exists \rightarrow x$ and $x:B, s:\{B\}$, then $x:B-w$ and $s:\{B-w\}$ if not already $s:\{B-s\}$.

Data structure inference for the naive transformation is especially easy, because we can determine the object flow graph entirely from local bottom-up analysis of the subexpressions $T1,\ldots,T5$, which appear in expression (3). For example, $T1 = Y-S$, requires S to be strongly based, because a membership test on S would be needed to compute $T1$. $T3 = f[T2]$ requires $T3$ to be strongly based, because of a membership test on $T3$ in computing $T3$. $T4 = X \cup T3$ would require a membership test on either X, $T3$, or $T4$. This arbitrary choice indicates that unlike bases, we have no unique strong/weak basing property for data structures. Our inference algorithm comes up with the following data structures:

Base $A:\{t\}$
Base $I:\{\{A-w\}\}$
$X:\{A-w\}$
$f:\{[I-s, A-w]\}$
$s:\{A-s\}$
$T1:\{A-w\}$
$T2:\{I-w\}$
$T3:\{A-s\}$
$T4:\{A-s\}$
$T5:\{A-w\}$

which are illustrated in Figure 3.

Note that the best real-time simulation was possible for this example, in that no hashing was required.

7 Transformation to Ada

Once we obtain our data structure subtypes, we can rewrite these types more concretely for a language such as Ada. That is, pointer, array, and record types could be introduced. Then our default data structures (which support asymptotic times) could be improved to gain constant factors in time and space. A general improvement could be obtained by eliminating any unused pointers from our data structures. This sort of refinement would lead to a comprehensive assortment of data structures as are found in the first several chapters of [AHU74].

In this section we specialize the data types associated with the SML code for attribute closure, and transform these types together with the SML code into Ada. The following rules can be applied mechanically:

1. A base together with all its strongly-based objects becomes an array of records, and each based object corresponds to one component of the base record. If the base elements are of some discrete type, then the index of the base array can be that discrete type. Otherwise the base can be represented as an array of integers. These indices are used solely to access the base.

2. For a strongly-based subset which is used only for insertion, deletion and membership, the corresponding component type is BOOLEAN. If the subset is also used for iteration, the component type is the base index, and the set is chained as indicated in Figure 1. Initializations of strongly based objects require an iteration over the base.

3. A weakly-based subset is a linked list of base indices. Iterations over weakly based objects are list traversals.

As a result of applying these rules we obtain the Ada code in Appendix C.

8 Implementation

Two implementations have been under developement, one within Mentor/Typol [DGDFJ87,Kah87,Des84] and another within the RAPTS transformational system [PH87,CP89]. Both systems use pattern directed first order inductive definitions to do semantic analysis. We already have some encouraging preliminary timing studies. In Figure 4 comparative benchmark timings (on a SUN 3/50) that compare our data structure transformations with two other data structure design algorithms are presented. The graph labeled SETL is low level SETL code [SDDS86] generated automatically in RAPTS using fixed point and finite differencing transformations. The graph labeled SETL-to-Ada represents Ada produced automatically by Doberkat's and Gutenbeil's system [DG87] applied to the RAPTS generated SETL. Ada1 represents Ada code manually generated by applying the data structure design method described here to the RAPTS generated code. For the graph reachability problem we also have a graph labeled Ada2, which is manually composed Ada that implements the same rough strategy but is not otherwise constrained.

9 Conclusion

Our approach to automatic data structure selection differs from other work in significant ways. Earlier work in the Artificial Intelligence community (e.g., [Low74,Bar79,DG87,Rov77,Kan81]) considered a wider assortment of data structures than us, but they relied on weaker heuristic methods in their 'expert systems'. For the most part, those methods were also more localized in their focus on successive refinement of a data structure for a single variable. The work that comes closest to ours is the SETL data structure selection and aggregation by basings[Sch75b,SSS81,DGC$^+$79,FSS83]. We have been motivated by an interest in overcoming some of the practical shortcomings in the design of the SETL optimizer, and in improving the theoretical underpinnings of data structure selection, especially with regard to complexity guarantees.

The work reported here is preliminary, but highly encouraging. We are currently working on several improvements and appications. The data structure design method is being generalized by adding more primitive operations (as are found in Chapter one of Tarjan's book[Tar87]) into SML. Dynamic bases and runtime data structure reorganization are also being investigated.

We have only provided some meta rules for data structure inference, and provided only two acceptable transformations - one based purely on operational semantics and one improved by finite differencing. The ad hoc approach used in [PH87], where attribute closure was also considered, used finite differencing to generate set machine code one order of magnitude faster than the code produced by our naive method.

References

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AHU83] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[Bar79] D. Barstow. *Knowledge-Based Program Construction*. Elsevier North-Holland, 1979.

[CC79] P. Cousot and R. Cousot. Constructive versions of tarski's fixed point theorems. *Pacific J. Math*, 82(1):43–57, May 1979.

[CP89] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, 1988/89.

[Cur69] H. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.

[Dat82] C. Date. *Introduction to Database Systems, Vol II*. Addison-Wesley, 1982.

[Des84] T. Despeyroux. Executable specification of static semantics. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer Verlag, 1984.

[DG87] E. Doberkat and U. Gutenbeil. Setl to ada tree transformations applied. *Information and Software Technology*, 29(10):548–557, Dec. 87.

[DGC+79] R. Dewar, A. Grand, Liu S. C., J. T. Schwartz, and E. Schonberg. Program by refinement, as exemplified by the setl representation sublanguage. *TOPLAS*, 1(1):27–49, July 1979.

[DGDFJ87] V. Donzeau-Gouge, C. Dubois, P. Facon, and F. Jean. Development of a programming environment for setl. SED Project Report, 1987.

[FM88] Y. Fuh and P. Mishra. Type inference with subtypes. In *Proc. 2nd European Symp. on Programming*, pages 94–114. Springer-Verlag, 1988. Lecture Notes in Computer Science 300.

[Fre87] K. Frenkel. An interview with the 1986 a. m. turing award recipients - john e. hopcroft and robert e. tarjan. *CACM*, 30(3):214–223, Mar 1987.

[FSS83] S. Freudenberger, J. T. Schwartz, and M. Sharir. Experience with the setl optimizer. *ACM TOPLAS*, 5(1):26–45, 1983.

[Gur84] Y. Gurevich. Toward logic tailored for computational complexity. In *Lecture Notes in Math: Computation and Proof Theory*, volume 1104, pages 175–216, 1984.

[Hin69] R. Hindley. The principal type-scheme of an object in Combinatory Logic. *Trans. Amer. Math. Soc.*, 146:29–60, Dec. 1969.

[Kah87] G. Kahn. Natural semantics. In *Lecture Notes in Computer Science 247*. Springer Verlag, 1987. Proc. STACS 1987.

[Kan81] E. Kant. *Efficiency in Program Synthesis*. UMI Research Press, 1981.

[Knu72] D. Knuth. *The Art of Computer Programming, 3 Volumes*. Addison-Wesley, 1968-1972.

[Lei83] D. Leivant. Polymorphic type inference. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, pages 88–98. ACM, Jan. 1983.

[Low74] J. Low. *Automatic Coding: Choice of Data Structures*. PhD thesis, Stanford University, Aug 1974.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[Mit83] J. Mitchell. Coercion and type inference. In *POPL*, 1983.

[PH87] R. Paige and F. Henglein. Mechanical translation of set theoretic problem specifications into efficient ram code - a case study. *Journal of Symbolic Computation*, 4(2):207–232, Aug. 1987.

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM TOPLAS*, 4(3):402–454, July 1982.

[PW78] M. Paterson and M. Wegman. Linear unification. *J. of Computer and System Sciences*, 16:158–167, 1978.

[Rov77] P. Rovner. Automatic representation selection for associative data structures. Technical Report TR10, Dept. of Computer Science, University of Rochester, 1977. Ph. D. Thesis, Harvard University.

[Sch75a] J. T. Schwartz. Optimization of very high level languages, parts i, ii. *J. of Computer Languages*, 1(2,3):161–218, 1975.

[Sch75b] J.T. Schwartz. Automatic data structure choice in a language of very high level. *CACM*, 18(12):722–728, Dec 1975.

[SDDS86] J. Schwartz, R. Dewar, D. Dubinsky, and E. Schonberg. *Programming with Sets: An introduction to SETL*. Springer-Verlag, 1986.

[SSS81] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in in setl programs. *ACM TOPLAS*, 3(2):126–143, Apr 1981.

[Tar87] R. Tarjan. Algorithm design. *CACM*, 30(3):204–213, Mar 1987.

[Ull80] J. Ullman. *Principles of Database Systems*. Computer Science Press, 1980.

[Wie83] G. Wiederhold. *Database Design*. McGraw-Hill, 1983.

A SML Code implementing Attribute Closure

```
                    S := {};
Loop:               T4 := {};
                    (for v ∈ x)
                        T4 with:= v;
                    end;
                    T2 := {};
                    (for y ∈ domain f)
                        T1 := {};
                        (for z ∈ y)
                            if z ∉ S then
                              T1 with:= z;
                            end;
                        end;
                        if T1 = {} then
                            T2 with:= y;
                        end;
                    end;
                    T3 := {};
                    (for w ∈ T2)
                        (for u ∈ f{w})
                            if u ∉ T3 then
                              T3 with:= u;
                            end;
                        end;
                    end;
                    (for v ∈ T3)
                        if v ∉ T4 then
                            T4 with:= v;
                        end;
                    end;
                    T5 := {};
                    (for a ∈ T4);
                        if a ∉ S then
                            T5 with:= a;
                        end;
                    end;
                    if ∃z ∈ T5 then
                        S with:= z;
                        goto loop;
                    end if
```

B A type system for $SQ2^+$

The (first-order) type expressions are all the expressions derivable from T in the following grammar:

```
------------------------------------------------------------------------

T       ::=     int | Bool | string |
                t1, t2, ... (type variables)
                [T, T] |
                { T }

------------------------------------------------------------------------
```

Note that function types, denoted by T $\mapsto$ T', respectively (T, T') $\mapsto$ T" below, are excluded from these type expressions. This has the consequence that functions cannot be passed to other functions; and they cannot be elements of tuples or sets. They are *not* first-class "citizens" in $SQ2^+$. A type system in the spirit of the Simply Typed Lambda Calculus is presented below. It is a specialization of the "patterned" form of presentation from [Lei83] and [FM88].

For all type expressions T, T1, T2, T', program expressions e, e1, e2,
and type assignments (= mappings from program variables to type expressions)
A (where A{x: T}(y) = A(y), if x $\neq$ y
T, if x = y)
the following are the axiom and deduction rule schemes of type system TT.

Axioms:

(VAR) $A\{x\colon T\} \vdash x\colon T$

Rules:

(i) abstractive rules:

(SET)
$$\frac{A\{x\colon T1\} \vdash e1\colon T2,\ e3\colon Bool,\ e2\colon \{T1\}}{A \vdash \{\ e1\colon x \in e2 \mid e3\ \}\colon \{\ T2\ \}}$$

(QUANT)
$$\frac{A\{x\colon T\} \vdash e2\colon Bool,\ e1\colon \{T\}}{A \vdash \forall\ x \in e1 \mid e2\colon Bool \quad A \vdash \exists\ x \in e1 \mid e2\colon Bool}$$

(FIX)
$$\frac{A\{x\colon T\} \vdash e\colon T}{A \vdash LFP\ x.\ e\colon T \quad A \vdash GFP\ x.\ e\colon T}$$

(ii) applicative rules:

(TUPL-INTR)
$$\frac{A \vdash e1\colon T1,\ e2\colon T2}{A \vdash [e1, e2]\colon [T1, T2]}$$

(TUPL-ELIM) $A \vdash e\colon [T1, T2]$

A ⊢ e.1: T1
A ⊢ e.2: T2

(CONST) nullop: T

A ⊢ nullop: T

(UNOP) A ⊢ e: T
unop: T ↦ T'

A ⊢ unop e: T'

(BINOP) A ⊢ e1: T1, e2: T2
binop: (T1, T2) ↦ T

A ⊢ e1 binop e2: T

Operator axioms:

domain: {[T1, T2]} ↦ {T1}
range: {[T1, T2]} ↦ {T2}
$.^{-1}$: {[T1, T2]} ↦ {[T2, T1]}
[set]: {T} ↦ int
.(.): ({[T1, T2]}, T1) ↦ T2
.{.}: ({[T1, T2]}, T1) ↦ {T2}
.[.]: ({[T1, T2]}, {T1}) ↦ {T2}
(∪): ({T}, {T}) ↦ {T}
(∩): ({T}, {T}) ↦ {T}
- [set]: ({T}, {T}) ↦ {T}
(⊆): ({T}, {T}) ↦ Bool
(∋): {T} ↦ T
- [int]: (int, int) ↦ int
+: (int, int) ↦ int
max, min: (int, int) ↦ int
<: (int, int) ↦ Bool
=: (T, T) ↦ Bool
(or): (Bool, Bool) ↦ Bool
(and): (Bool, Bool) ↦ Bool
(not): Bool ↦ Bool
(∈): (T, {T}) ↦ Bool
(Ω): T
{}: {T}
true, false: Bool
..., -1, 0, 1, ...: int
..., 'a', 'aa', ...: string

A formula of the form A ⊢ e: T is called a typing (statement). A program e is said to be typable (in TT) if there exist type assignment A and type expression T such that the typing A ⊢ e: T is derivable in TT. Given a straightforward denotational Strachey-Scott style semantics for SQ+ (a naive set-theoretic interpretation of domains will also do since there are no first-class functions), it is intuitively clear that this system is semantically sound. (See [Mil78] for an elaboration of semantic soundness). Since $SQ2^{+}$ has unrestricted computational power, it is clear that this system is semantically incomplete. Otherwise it would be undecidable (there is a linear-time decision calculus for TT). It is well-known [Hin69,Cur69] that this system has a principal typing property; that is, for every typable e there is a typing A ⊢ e: T such that for every derivable typing A' ⊢ e: T' there is a substitution (on type variables) such that `A'` = S(`A`)

and T' = S(T). Note that this system is not polymorphic in the sense that ML is since it lacks a 'let'-construct with its polymorphic typing rule.

C Ada Implementation of Attribute Closure

The variables used in our example receive the following declarations

```
type A_index is ...

type A_Rec is record
   S: boolean ;
   T3, T4: A_index ;
end record ;

A_base: array(A_index) of A_Rec;

type I_Rec is record
   I: A_list ;
   F: A_list ;
end record ;

I_Base: array(I_index) of I_Rec;
T4first, T3first: A_index;  -- to hold first element of set.
```

In addition, standard instantiations of a generic list package provide lists of A_index and lists of I_index, together with insert, delete, and newlist primitives.

```
    u, v, x, y, T1, T5: A_list ;
    w, T2: I_list ;


    -- S := {}
    for i in A_BASE'range loop A_BASE(i).S := false; end loop;


  <<loop>>
    -- T4 := {}
    for i in A_BASE'range loop A_BASE(i).T4 := null_index; end loop;
    T4first := null_index ;

    v := x ;
    if v /= null then
        T4first := v.index ;
        A_BASE(v.index).T4 := null_index ;
        v := v.next ;
        while v /= null loop
            A_BASE(v.index).T4 := T4first ;
            T4first := v;
            v := v.next ;
        end while ;
    end if ;

    T2 := new_list(I_index) ;
    for i in I_index'range loop
        y := I_BASE(i).f;
        T1 := new_list(A_index) ;
        z := y ;
        while z /= null loop
```

```
       if not A_BASE(z.index).S then insert(z.index, T1) ; end if ;
       z := z.next ;
    end while ;
    if T1 = null then insert(i, T2) ; end if ;
end loop;


-- Initialize T3.
for i in A_BASE'range loop A_BASE(i).T3 := null_index; end loop;
T3first := null_index ;

w := T2 ;
while w /= null loop
    u := I_BASE(w.index).f ;
    if u /= null then
       T3first := u.index ;
       A_BASE(u.index).T3 := null_index ;
       u := u.next ;
       while u /= null loop
           if A_BASE(u.index).T3 = null_index then -- chain element to T3.
               A_BASE(u.index).T3 := T3first ;
               T3first := u.index;
           end if ;
           u := u.next;
       end while ;
    end if ;
end while ;


v1 := T3first ;
while v1 /= null_index loop
    if A_BASE(v1).T4 = null_index then -- chain element to T4.
        A_BASE(v1).T4 := T4first ;
        T4first := v1;
    end if ;
    v1 := A_BASE(v1).T3 ; -- iterate through T3.
end loop;

T5 := new_list(A_index) ;
a := T4first;
while a /= null_index loop
   if not A_BASE(a).S then insert(a, T5) ; end if ;
   a := A_BASE(a).T4 ;
end loop ;

if T5 /= null then A_BASE(T5.index).S := true ; go to loop ; end if;
```

D Weakly and strongly based data structures

E Object flow graph

F Base declarations

G Timing studies