```
      T4 := v1;
      end if ;
      v1 := A_BASE(v1).T3 ;
                  -- iterate through T3.
end loop;
      T5 := new_list ;
a := T4;
while a /= last_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
                  -- insert first element in S
      A_BASE(T5.index).S := true ; goto start ;
end if;
-- Display result.
for i in A_BASE'range loop
      if A_BASE(i).S then put(i) ; put(" ") ; end if ;
end loop;

end attribute_closure;
```

```
T4 := last_index ;

v := X ;
while  v /= null loop
     A_BASE(v.index).T4 := T4 ;
     T4 := v.index;
     v := v.next ;
end loop ;

T2 := new_list ;
for i in I_index loop
     y := I_BASE(i).I;
     T1 := new_list ;
     z := y ;
     while z /= null loop
          if not A_BASE(z.index).S then insert(z.index, T1) ; end if ;
          z := z.next ;
     end loop ;
     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;

T3 := last_index ;

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

v1 := T3 ;
while v1 /= last_index loop
     if A_BASE(v1).T4 = null_index then
               -- chain element to T4.
     A_BASE(v1).T4 := T4 ;
```

```
type A_Rec is record
     S: boolean ;              -- strongly based subset.
     T3, T4 : A_index ;        -- strongly based subsets, with iteration.
end record ;
A_base: array(A_index) of A_Rec;
type I_Rec is record
     I: A_list ;               -- domain,
     F: A_list ;               -- and range of function f,
end record ;
I_Base: array(I_index) of I_Rec;
X, T1, T5: A_list;
T2: I_list;
T4, T3: A_index;               -- to hold first element of set.
u, v, y, z: A_list;            -- bound variables in iterators.
a, v1: A_index ;               -- ditto.
w: I_list ;
package int_io is new integer_io(A_index) ; use int_io ;
function get_A_list return A_list is
     -- To initialize X and F. Input is a list of indices, without duplicates
     l: A_list ;
     e: A_index ;
begin
     l:= new_list ;
     get(e) ;
     while e /= null_index loop
          insert(e, l);
          get(e) ;
     end loop ;
     return l ;
end get_A_list ;
begin

X := get_A_list;
-- Input F, as list of pairs of lists, and build I_BASE.
for i in I_index loop
     I_BASE(i).I := get_A_list ;
     exit when I_BASE(i).I = null;
     I_BASE(i).F := get_A_list ;
end loop ;

-- S := {} S is strongly based.
for i in A_BASE'range loop A_BASE(i).S := false; end loop;
     <<start>>
-- T4 := {}
for i in A_BASE'range loop
     A_BASE(i).T4 := null_index;
end loop;
```

bases

$I = \mathbf{domain}\, F$

$A = (\bigcup \mathbf{domain}\, F) \cup X \cup \mathbf{range}\, F$

based variables

$S : \mathrm{set}(A)_{a+}$

$T4 : \mathrm{set}(A)_{ar+}$

$v : A$

$X : \mathrm{set}(A)_{r}$

$T2 : \mathrm{set}(I)_{r+}$

$y : I$

$z : A$

$F : \mathrm{set}(I, A)_{r,r}$

$T1 : \mathrm{set}(A)_{r+}$

$w : I$

$u : A$

$T3 : \mathrm{set}(A)_{ar+}$

$T5 : \mathrm{set}(A)_{r+}$

$a : A$

E Ada code for attribute closure

```
with text_io; use text_io;
with lists;
    -- a generic package that provides the functions:
    -- insert, delete, and new_list.
procedure attribute_closure is
    total_attributes: constant := 50;
    total_relations : constant := 50;

    type A_index is range 0 .. total_attributes;
    type I_index is range 0 .. total_relations;
    null_index: constant A_index := 0 ;
    last_index: constant A_index := A_index'last ; -- to mark end of lists
    package A_lists is new lists(A_index) ; use A_lists ;
    package I_lists is new lists(I_index) ; use I_lists ;
    subtype A_list is A_lists.list ;
    subtype I_list is I_lists.list ;
    -- S is strongly based and only used for membership and insertion: it
    -- is encoded as a boolean attached to each base value.
    -- T3 and T4 are strongly based and used for membership, insertion, and
    -- iteration. They can be encoded by a boolean value and a link, or else
    -- (at the price of some extra coding) by a circular list and a pointer to
    -- the first element. The code below uses a list with two special pointers:
    -- null_index to designate the absence of an element, and last_index, used
    -- only on the last element of the list.
```

```
        ( for z: A ∈ y)
            if z ∉ S then
                T1 with:= z;
            end;
        end;
        if T1 = {} then
            T2 with:= y;
        end;
    end;
    T3 := {}; –maintain T3
    (for w: I ∈ T2)
        ( for u: A ∈ F{w})
            if u ∉ T3 then
                T3 with:= u;
            end;
        end;
    end;
    T4 := {}; –maintain T4
    (for v: A ∈ X)
        T4 with:= v;
    end;
    (for v: A ∈ T3)
        if v ∉ T4 then
            T4 with:= v;
        end;
    end;
    T5 := {}; –maintain T5
    (for a: A ∈ T4)
        if a ∉ S then
            T5 with:= a;
        end;
    end;
    if ∃ z ∈ T5 | true then
        S with:= z;
        goto start;
    end if;
    print('closure: ', S) ;

end program attribute_closure ;
```

The type assignments for all variables in this code are listed below. The subscripts indicate whether access (a), retrieval (r), or addition ($+$) are performed on them. Consequently, $S, T3$ and $T4$ will be strongly based on A. In fact, since there is no retrieval operation performed on S it can be implemented as a bit field associated with the base A. The domain of F and $T2$ will be weakly based on I. Finally, the image sets of F under every domain point, X, $T1$, and $T5$ will be weakly based on A.

type specification QUERIES is

(FUNC-EVAL) $\dfrac{f ::< \tau, \tau' > \quad e : \tau}{fe : \tau'}$

(FUNC-BIND) $\dfrac{f ::< \tau, \tau' > \vdash e : \tau'' \quad x : \tau \vdash e' : \tau'}{fx = e'; e : \tau''}$

(FIXDEF) $\dfrac{f : \tau \quad x : \tau \vdash e : \tau \quad x : \tau \vdash e' : \tau'}{x_f = e; e' : \tau'}$

C Subtype inference system for SQ2$^+$

The subtype inference system for SQ2$^+$adds the following axiom and rule schemes to the type inference system of Appendix B.

type specification COERCIONS is

(REFL) $\tau \leq \tau$

(TRANS) $\dfrac{\tau \leq \tau' \quad \tau' \leq \tau''}{\tau \leq \tau''}$

(PAIR-COMP) $\dfrac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{(\tau_1, \tau_2) \leq (\tau_1', \tau_2')}$

(SET-COMP) $\dfrac{\tau \leq \tau'}{\mathrm{set}(\tau) \leq \mathrm{set}(\tau')}$

(COERCE) $\dfrac{e : \tau \quad \tau \leq \tau'}{e : \tau'}$

D SETM code for attribute closure

```
program attribute_closure ;
read(X);
read(F) ;
S := {};
start: T2 := {}; –maintain T2
     (for y: [I ¡ {A}] ∈ domain F)
          T1 := {}; –maintain T1
```

type specification BOOL is

(TRUE) $\quad \mathbf{true} : \mathbf{bool}$

(FALSE) $\quad \mathbf{false} : \mathbf{bool}$

(COND) $\quad \dfrac{e : \mathbf{bool} \quad e' : \tau \quad e'' : \tau}{\mathbf{if}\, e\, \mathbf{then}\, e'\, \mathbf{else}\, e'' : \tau}$

type specification INTEGER is
using BOOL
...

type specification PAIR is

(PAIR) $\quad \dfrac{e : \tau \quad e' : \tau'}{(e, e') : (\tau, \tau')}$

(PROJ-I) $\quad \dfrac{e : (\tau, \tau')}{e.1 : \tau}$

(PROJ-II) $\quad \dfrac{e : (\tau, \tau')}{e.2 : \tau'}$

type specification SET is
using BOOL
with $(\forall S, x, y)\, S, x, y = S, y, x$

(EMPTYSET) $\quad \{\} : \mathrm{set}(\tau)$

(INSERT) $\quad \dfrac{M : \mathrm{set}(\tau) \quad e : \tau}{M, e : \mathrm{set}(\tau)}$

(SETITER) $\quad \dfrac{e' : \tau' \quad M : \mathrm{set}(\tau) \quad x : \tau \vdash P : \mathbf{bool} \quad x : \tau \vdash e : \tau' \quad x' : \tau', x'' : \tau' \vdash x' \oplus x'' : \tau'}{e' \oplus e : x \in M \mid P : \tau'}$

semantic specification SET is
using BOOL
with $(\forall S, x, y)\ S, x, y = S, y, x$

(EMPTYSET) $\{\} \to \{\} : \mathrm{set}(\tau)$

(INSERT-I)
$$\frac{M \to S : \mathrm{set}(\tau) \quad e \to v : \tau \quad S = T, v \text{ for some } T}{M \text{ with } e \to S : \mathrm{set}(\tau)}$$

(INSERT-II)
$$\frac{M \to S : \mathrm{set}(\tau) \quad e \to v : \tau \quad S \neq T, v \text{ for all } T}{M \text{ with } e \to S, v : \mathrm{set}(\tau)}$$

(SETITER-I)
$$\frac{e' \to v' : \tau' \quad M \to \{\} : \mathrm{set}(\tau)}{e' \oplus e : x \in M \mid P \to v' : \tau'}$$

(SETITER-II)
$$\frac{\begin{array}{l} e' \to v' : \tau' \\ M \to S, w : \mathrm{set}(\tau) \\ x \to w : \tau \vdash P \to \mathbf{true} : \mathbf{bool} \\ x \to w : \tau \vdash e \to v : \tau' \\ x' \to v' : \tau', x'' \to v : \tau' \vdash x' \oplus x'' \to v'' : \tau' \\ y \to v'' : \tau', z \to S : \mathrm{set}(\tau) \vdash y \oplus e : x \in z \mid P \to v''' : \tau' \end{array}}{e' \oplus e : x \in M \mid P \to v''' : \tau'}$$

(SETITER-III)
$$\frac{\begin{array}{l} e' \to v' : \tau' \\ M \to S, w : \mathrm{set}(\tau) \\ x \to w : \tau \vdash P \to \mathbf{false} : \mathbf{bool} \\ y \to v' : \tau', z \to S : \mathrm{set}(\tau) \vdash y \oplus e : x \in z \mid P \to v'' : \tau' \end{array}}{e' \oplus e : x \in M \mid P \to v'' : \tau'}$$

B Type inference system for SQ2$^+$

Following is our natural deduction style type inference system that defines a strongly typed subset of SQ2$^+$. The typing rules are for the kernel of SQ2$^+$in Appendix A. Constructs outside the kernel are typed by typing their definitions in terms of the kernel language, which results in derived typing rules not presented here.

semantic specification BOOL is

(TRUE) $\quad \mathbf{true} \to \mathbf{true} : \mathbf{bool}$

(FALSE) $\quad \mathbf{false} \to \mathbf{false} : \mathbf{bool}$

(COND-I) $\quad \dfrac{e \to \mathbf{true} : \mathbf{bool} \quad e' \to v' : \tau}{\mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'' \to v' : \tau}$

(COND-II) $\quad \dfrac{e \to \mathbf{false} : \mathbf{bool} \quad e'' \to v'' : \tau}{\mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'' \to v'' : \tau}$

semantic specification INTEGER is
using BOOL
...

semantic specification PAIR is

(PAIR) $\quad \dfrac{e \to v : \tau \quad e' \to v' : \tau'}{(e, e') \to (v, v') : (\tau, \tau')}$

(PROJ-I) $\quad \dfrac{e \to (v, v') : (\tau, \tau')}{e.1 \to v : \tau}$

(PROJ-II) $\quad \dfrac{e \to (v, v') : (\tau, \tau')}{e.2 \to v' : \tau'}$

insertion operator "," satisfies the equality $S, x, y = S, y, x$ for all S, x, y. Since the semantics of element insertion guarantees that no value is inserted twice into a set we can identify the formal value terms $\{\}, v_1, \ldots, v_k$ with finite sets and write the more familiar $\{v_1, \ldots, v_k\}$.

Much of the expressive power of set queries is provided by the generic *set iterator* construct $e' \oplus e : x \in M \mid P$. Here e, e', M and P are expressions, x is an identifier, and $\oplus$ is a (predefined) binary operator or a (user-defined) binary function. If $P = \mathbf{true}$ then "$\mid P$" may be omitted.

Using set iteration the set cardinality operator can be defined by $|M| = 0 + 1 : x \in M \mid \mathbf{true}$. Similarly, $M \cup N$ is defined by $M, x : x \in N$. Finally, *set comprehension* is $\{\}, e : x \in M \mid P$, which we write $\{e : x \in M \mid P\}$ in concrete syntax. Set comprehension with two nested iterators $\{e : x \in M, y \in N \mid P\}$ where x may occur free in N is defined by $\{\} \cup \{e : y \in N \mid P\} : x \in M$; set comprehensions with an arbitrary number of nested iterators may be defined analogously. It is not possible to define set iteration in terms of set comprehension and set reduction since set iteration does *not* eliminate duplicate values computed for e before applying the operator $\oplus$ in $e' \oplus e : x \in M \mid P$ (see rules (SETITER-II) and (SETITER-III) below), but set comprehension *does*.

The operational semantics of our language is given using syntax-directed (structural) inference rules for formal statements $e \rightarrow v : \tau$, which relate expressions to their values. Since we do not treat functions as values we also provide formal (closure) binding statements $f ::< x, e >$, which may be used as (hypothetical) assumptions in typing derivations. An expression e has value v if there is a natural deduction style proof of $e \rightarrow v$ using the (substitution instances of the) following axioms and inference rules.

semantic specification QUERIES is

(FUNC-EVAL)
$$\frac{f ::< x : \tau, e' : \tau' > \quad e \rightarrow v : \tau \quad x \rightarrow v : \tau \vdash e' \rightarrow v' : \tau'}{fe \rightarrow v' : \tau'}$$

(FUNC-BIND)
$$\frac{f ::< x : \tau, e' : \tau' > \vdash e \rightarrow v : \tau''}{fx = e'; e \rightarrow v : \tau''}$$

(FIXDEF-I)
$$\frac{f \rightarrow v : \tau \quad x \rightarrow v : \tau \vdash e \rightarrow v : \tau \quad x \rightarrow v : \tau \vdash e' \rightarrow v' : \tau'}{x_f = e; e' \rightarrow v' : \tau'}$$

(FIXDEF-II)
$$\frac{f \rightarrow v : \tau \quad x \rightarrow v : \tau \vdash e \rightarrow v' : \tau \quad y \rightarrow v' \vdash x_y = e; e' \rightarrow v'' : \tau'}{x_f = e; e' \rightarrow v'' : \tau'}$$

[Pai89] R. Paige. Real-time simulation of a set machine on a RAM. In R. Janicki and W. Koczkodaj, editors, *Computing and Information, Vol II*, pages 69–73, May 1989. ICCI 89.

[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.

[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. Schwartz. Optimization of very high level languages – I: Value transmission and its corollaries. *J. Computer Languages*, 1:161–194, 1975.

[Sch75b] J. Schwartz. Optimization of very high level languages – II: Deducing relationships of inclusion and membership. *J. Computer Languages*, 1:197–218, 1975.

[Sch75c] J.T. Schwartz. Automatic data structure choice in a language of very high level. *CACM*, 18(12):722–728, Dec 1975.

[Set89] R. Sethi. *Programming Languages: Concepts and Constructs.* Addison-Wesley, 1989.

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

[Tar87] R. Tarjan. Algorithm design. *CACM*, 30(3):204–213, Mar 1987. 1986 ACM Turing Award Lecture.

[Ull82] J. Ullman. *Principles of Database Systems.* Computer Science Press, 2nd edition, 1982.

[Wan87] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987.

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

A Structural operational semantics of SQ2$^+$

We provide a formal semantics for a *kernel* of SQ2$^+$. SQ2$^+$constructs and operators outside the kernel can be syntactically defined in terms of the kernel primitives.

The primitive set constructors are the empty set, $\{\}$, and element addition, written M, x where M is a set and x an element. To capture the "data" nondeterminism of sets the element

[For82] C. Forgy. Rete, a fast algorithm for the many patterns many objects match problem. *Artificial Intelligence*, 19(3):17–37, 1982.

[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. Schwartz, and M. Sharir. Experience with the SETL optimizer. *ACM TOPLAS*, 5(1):26–45, Jan. 1983.

[GLT89] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[Hen88] F. Henglein. Type inference and semi-unification. In *Proc. ACM Conf. on LISP and Functional Programming (LFP), Snowbird, Utah*. ACM, ACM Press, July 1988.

[Hen91a] F. Henglein. Dynamic typing. Unpublished manuscript, March 1991.

[Hen91b] F. Henglein. Efficient type inference for higher-order binding-time analysis. Submitted for publication, Feb. 1991.

[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 *Proc. STACS'87*. Springer-Verlag, 1987. Lecture Notes in Computer Science, Vol. 247.

[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.

[KTU90] A. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is DEXPTIME-complete. In *Proc. 15th Coll. on Trees in Algebra and Programming (CAAP), Copenhagen, Denmark*, pages 206–220. Springer, May 1990. Lecture Notes in Computer Science, Vol. 431.

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

[Mai90] H. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proc. POPL '90*. ACM, Jan. 1990.

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

[Mit84] J. Mitchell. Coercion and type inference. In *Proc. 11th ACM Symp. on Principles of Programming Languages (POPL)*, 1984.

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

[BG91] L. Blaine and A. Goldberg. Dtre - a semi-automatic transformation system. In B. Moeller, editor, *Constructing Programs from Specifications.* North-Holland, 1991.

[CP89] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, 1989.

[CP91] J. Cai and R. Paige. Look Ma, no hashing, and no arrays neither. In *Proc. 18th Annual ACM Symp. on Principles of Programming Languages (POPL), Orlando, Florida*, pages 143–154, Jan. 1991.

[CPT90] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoretical Computer Science*, 1990. to appear; also in Proc. CAAP '90, LNCS 431, pp. 72-86.

[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.

[DG87] E. Doberkat and U. Gutenbeil. SETL to Ada — tree transformations applied. *Information and Software Technology*, 29(10):548–557, Dec. 1987.

[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.

[DGHKL84] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: The Mentor experience. In D. Barstow, H. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments.* McGraw-Hill, 1984.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[Ede85] E. Eder. Properties of substitutions and unifications. *J. Symbolic Computation*, 1:31–46, 1985.

[FHR90] P. Facon and N. Hadj-Rabia. Traduction optimisee d'un langage ensembliste en ada. In *3e JIGLA, Toulouse, France*, Dec. 1990.

[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.

9 Conclusion

Our approach to automatic data structure selection differs from other work in significant ways. Earlier work in the Artificial Intelligence community considered a wider assortment of data structures than us, but they relied on weaker heuristic methods drawing on expert systems technology (e.g., [Low74, Bar79, DG87, Rov77, Kan81]). 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[Sch75c, 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 implementation of top-down subtype analysis and data structure selection is ongoing as part of the RAPTS program transformation project. Another recent ongoing implementation effort is reported in [FHR90].

The theoretical work that comes closest to our simulation approach is due to Ben-Amram and Galil [BAG91], who give both upper and lower bounds for simulating any RAM (with array processing) program of time t and space s on a Pointer Machine in $\theta(t\mathbf{log}s)$ time. Their RAM and Pointer Machine models are similar to the ones discussed in [Pai89], where sufficient conditions were presented for simulating a RAM on a Pointer Machine in real-time. That real-time simulation technique is similar to the set machine simulation technique just discussed.

The work reported here is preliminary, but highly encouraging. We are currently working on several improvements and applications. The data structure design method is being generalized by adding more primitive operations (as are found in Chapter 1 of Tarjan's book[Tar87]) into SETM. Dynamic bases and runtime data structure reorganization are also being investigated. Finally, it would be useful to formally verify our data structure selection method using a system such as DTRE[BG91].

Acknowledgments

We would like to thank Ulrich Gutenbeil for translating the SETM code of Appendix D into Ada using the Doberkat/Gutenbeil SETL-to-Ada translator and for providing us with the Ada sources of their run-time system. This facilitated the empirical comparisons of Section 8.

References

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

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

[BAG91] A. Ben-Amram and Z. Galil. On pointers versus addresses. *JACM*, 1991. to appear; also in 29th IEEE FOCS, 1988, pp. 532-538.

The run-time environments of these four versions are sufficiently different to make their analysis delicate. Neither Ada version has a garbage collector, but the Essen library uses allocators systematically, without any storage reclamation (explicit deallocation in the Ada1 code could be added with little effort, leading to much better memory usage). Both SETL versions are interpreted, while the Ada versions are compiled.

Most of the code for Ada1 consists of subprogram calls to the run-time library, so that it mimics closely the interpreted code of the SETL versions. As a result, the relative performance of SETL and Ada1 reflects the relative sophistication of their respective run-time libraries. The data structures of the Essen library are simpler than those of SETL (e.g., hash-tables are of fixed size and contain separate sub-tables for various element types) and thus perform worse than those of the SETL library for any but the smallest inputs. For programs with very dynamic and short-lived set values, the Essen run-time support, remarkable a coding effort as it is, suffers from its choice of suboptimal data structures and its absence of integration with a storage manager.

The original purpose of subtypes and based data structures was to minimize associative access to sets and mappings, by replacing associative operations (hash-table retrievals) with faster indexing and dereferencing operations. It is nevertheless clear that the gains in efficiency achieved by subtypes are more far-reaching.

- Operations on based structures are simpler than those on standard sets and maps, and code for these operations can be emitted in-line, both because code for the operations themselves is shorter, and because run-time inspection of type-tags is unnecessary. Thus the use of subtypes makes true compilation of the code possible.

- The SQ2$^+$specification guarantees that the base sets are fully determined from input, so that bases can be allocated quasi-statically (for example, as local variables to an outer block). Similarly, strongly-based sets and maps are stored as attributes of base elements, and generate no additional storage allocation actions; only weakly based sets and maps need to be dynamically allocated. By minimizing storage management activity, subtypes suppress the largest overhead associated with modifications of the store in high-level languages.

For all programs that manipulate sets and maps, the two advantages of true compilation and minimal heap usage will provide a better-than-constant factor improvement in run-time performance. This is comparable to the improvements quoted for compiled LISP. What is notable is that this improvement is obtained for programs with set constructs, whose semantic level is much higher than list-based programs, and for which conventional optimization techniques alone do not yield better run-time performance.
One final note: conventional optimization techniques still play a vital role in our system. For example, the transformation for SQ2$^+$to SETL depends on substantial live-dead analysis and dead-code elimination. The Ada1 version would display better performance and memory utilization if explicit deallocation were used for dead values (all set temporaries in the main loop) but automatic generation of deallocation operations is a variant of copy optimization performed on the SETL code. Based representations allow us to make best use of these conventional optimization methods, and in so doing they capture aspects of manual coding that have escaped so far any attempts at mechanization.

Attributes (N)	50	100	200	400	500
Input size	40	155	482	1640	2504
SETL	0.32	2.26	9.76	67.70	123.58
SETL2	0.1	0.5	2.3	16.6	31.4
Ada1	0.36	1.35	6.29		
Ada2	0.08	0.18	0.90	4.43	8.65

Table 2: Attribute closure run-times for random dependencies

Attributes (N)	50	100	200	250	300
Input size	176	645	2108	3006	4289
SETL	7.4	41.3	254.3	467.6	725.9
SETL2	1.8	10.0	57.7	104.2	169.0
Ada1	16.6	56.2	868.7		
Ada2	0.58	2.07	11.09	24.09	36.80

Table 3: Attribute closure run-times for "dense" dependencies

an attribute). Let N be the number of distinct attributes stored F, and let the input size be the total number of attributes stored in all of the attribute dependencies in F; i.e.,

$$\begin{aligned} N &= |(\bigcup \mathbf{domain}\ F) \cup X \cup \mathbf{range}\ F| \\ \text{input size} &= \sum_{x \in \mathbf{domain}\ F} (|x| + 1) \end{aligned}$$

We ran these four implementations for different values of N, input sizes, and characteristics. The runs were made on an empty SUN3/160 server, running SunOS 4.1. The results of our experiments are summarized in Tables 2 and 3. All run-time figures are in seconds, corresponding to clock time in Ada and the time function in SETL. The times for SETL2 are user times obtained from the Unix time function. System time usually adds 10%, so that its inclusion does not affect the comparisons significantly. Table 2 gives the results for randomly generated functional dependencies with the following characteristics. The initial set X has cardinality $c = N/10$, and F has cardinality $2c$. Half of the functional dependencies $[S, a] \in F$ satisfy $S \subseteq X$, and half of them do not. Table 3 gives the results for dense sets of functional dependencies, where the attribute closure contains all N attributes.

The results of both experiments are consistent:

- The SETL2 version is about 3-4 times faster than SETL.
- Ada1 is unstable and loops or raises storage_error for the larger inputs.
- Ada2 is 4–5 times faster than SETL2.

guarantee all but the base conditions. We can say the same for the more efficient finite differencing transformation described in [PH87]. These transformations will also guarantee the base conditions for SETM code generated from an SQ2$^+$program e if the types of the subexpressions of e satisfy certain conditions. If B is a type variable then several of these conditions are listed below:

- $S \cup T : \mathrm{set}(B)$
- $\{e : x \in S | k\} : \mathrm{set}(B)$
- $x_f = e_1 : \mathrm{set}(B); e_2$
- $x : B \in S : \mathrm{set}(B)$

If these easily checkable typing conditions are satisfied for an SQ2$^+$program, then real-time simulation is achievable using the following simple heuristic for selecting weakly or strongly based representations. (1) All sets undergoing associative access are strongly based. (2) All other sets that must satisfy base compatibility conditions are weakly based. (3) All other sets are unbased.

This heuristic has been used to obtain the final representations for all variables occurring in the SETM program in Appendix D. The results are listed at the bottom of the code in Appendix D. The SETM code, together with the subtypings, is finally translated into pointer structures and pointer operations in a suitably low-level language. In our example this is Ada, shown in Appendix E. For reasons of efficiency the code uses arrays as well as lists. Note that the Ada code expects a slightly different input format from the SETM code. It first computes the two bases, I and A, and initializes all variables. Then it simulates every SETM step using the chosen data structures.

8 Empirical results

We tested 4 implementations of the same attribute closure algorithm:

- The SETL code obtained from the SQ2$^+$specification by naive finite differencing given in Appendix D.
- The same procedure transcribed into SETL2, a new implementation of SETL written in C.
- Ada1, a direct transcription of the same procedure into Ada, using the library of set primitives written by Doberkat and Gutenbeil [DG87].
- Ada2, a hand-coded version written in Ada and given in Appendix E, which uses the subtypes inferred by our algorithm as described in Appendix D. Both Ada versions were compiled with the Alsys compiler.

Recall that the attribute closure problem inputs a set X of the attributes and a set F of functional dependencies (each of the form $[S, a]$, where S is a set of attributes and a is

```
        gen(x ∪ m, n);),
    enter(type,[m,set(t)]),
    where m = newvar.
if match gen(x ∪ y: set(t), n), type(x ∪ y, set(t)), is-var(x),
    is-var(y) then /* both x and y are variables */
    rewrite(
        n := {};
        (for z: t in x)
            n with:= z;
        end;
        (for z in y)
            n with:= z;
        end;),
    where z = newvar.
```

In [Pai89] sufficient semantic conditions were given to guarantee a real-time simulation of SETM code on a RAM. Three kinds of conditions are mentioned. First are the conditions that guarantee a copy/value semantics with an efficient pointer implementation; i.e.,

- If S and T are sets, then assignments $S := T$ and S **with** $:= T$ are implemented by copying a pointer to the value of T.
- Element addition S **with** $:= x$ and deletion S **less** $:= x$ are destructive to the body of S. Just before either of these operations is performed, there must be no live pointers to the value of S.

Second, there are several base conditions. Consider an 'object flow graph' (OFG) that abstracts potential interactions between values in a SETM program at runtime. For each retrieval operation in an SETM program, where an object x is retrieved from a set s, we draw an edge $s\exists \rightarrow x$; for each SETM 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 OFG is related to but simpler than Schwartz's value flow analysis [Sch75a, Sch75b]. Given an object flow graph for a typed SETM program, we require the base conditions given below (where t is a type variable).

- For each variable $s : \mathrm{set}(t)$ and each OFG edge $x+ \rightarrow s$, we have $x : t$.
- For each variable $x : t$ and each OFG edge $s\exists \rightarrow x$, we have $s : \mathrm{set}(t)$.
- For each OFG edge $x \in\!\!\rightarrow s$, we have $x : t$ and $s : \mathrm{set}(t)$.

Finally, we assume that just before every element addition S **with** $:= x$, x does not belong to S. Likewise, just before every element deletion S **less** $:= x$, x belongs to S. If all these conditions can be met then the SETM code can be simulated in real-time on a RAM in accordance with the complexities given in Table 1.

The fixed point transformation and the naive finite difference transformation implemented (according to operational semantics of SQ2$^+$) in RAPTS and illustrated here

```
/* This rewrite rule replaces the outermost SQ2+query e by the result
of gencode, and is applied only once. */
    if match e, type(e,t) then
        rewrite(
            gen(e, n: t); /* evaluate e and store value in n */
            print(n);),
        where n = newvar.
/* fixed point iteration for set type */
    if match gen(Xe1 = e2; e3, n), type(Xe1,set(t)) then
        rewrite(
            gen(e1, X);
        L:  gen(e2 - X, w:);
            if exists z in w then
                X with:= z;
                goto L;
            end;
            gen(e3, n);),
        enter(type,[e2 - X: set(t)]), enter(type,[w,set(t)]), enter(type,[z,t]),
        where m = newvar, z = newvar, w = newvar, L = newvar.
/* set comprehension */
    if match gen({e1: x ∈ e2 | e3}, n), type(e1,t1), type(e2,set(t2)),
        type(e3,bool) then
        rewrite(
            n := {};
            gen(e2, m);
            (for x in m)
                gen(e3, p);
                if p then
                    gen(e1, q);
                    n with:= q;
                end;
            end;),
        enter(type,[m,set(t2)]),enter(type,[p,bool]), enter(type,[q,t1]),
        where m = newvar, p = newvar, q = newvar.
/* set union */
    if match gen(x ∪ y, n), type(x ∪ y,set(t)), not is-var(x) then
/* x is not a variable */
        rewrite(
            gen(x, m);
            gen(m ∪ y, n);),
        enter(type,[m,set(t)]),
        where m = newvar.
    if match gen(x ∪ y, n), type(x ∪ y, set(t)), not is-var(y) then
        rewrite(
            gen(y, m);
```

Example: (Transformation of attribute closure) The principal simple subtyping derivation of Section 5 for the SQ2$^+$attribute closure query can be transformed along with the query itself. Specifically, for the type containment set $\{I \leq \text{set}(A)\}$ and the type assignment $\{X : \text{set}(A), F : \text{set}(I, A)\}$ the fixed point transformation detailed in [CP89] turns the attribute closure query (with D unfolded in the code) into

$S : \text{set}(A) := \{\}$;
(**while** $\exists(z : A) \in (X \cup \{a : (Y, a) \in F \mid Y \subseteq S\} - S)$)
    S with $:= z$;
end while;

Finite differencing, in its simplest form, keeps the values of subexpressions e in separate variables T and maintains invariants $T = e$. Thus, the above code can be transformed into typed SETL,

$S := \{\}$;
maintain
    $T1 : \text{set}(A) = [I \leq \text{set}(A)]Y - S$
    $T2 : \text{set}(I) = \{Y : (Y : I) \in$ **domain** $F \mid T1 = \{\}\}$
    $T3 : \text{set}(A) = \{a : (Y : I, a : A) \in F \mid Y \in T2\}$
    $T4 : \text{set}(A) = X \cup T3$
    $T5 : \text{set}(A) = T4 - S$
in
    (**while** $\exists z \in T5$)
        S with $:= z$;
    end while;
end maintain;

where the scope of each invariant $T = e$ is defined by the place where variable T is used. This program is then transformed by naive finite differencing into the typed SETM code appearing in Appendix D. Subtypes for bound variables appearing in Appendix D are derived easily from the subtypes of the variables shown above. ■

Below is a fragment of the rewriting rules implemented in RAPTS for compiling SQ2$^+$queries into SETM code. More efficient code can be generated using the finite differencing transformations described in [PH87]. Recall that these rules are applied to a given program P, and that a rule can be applied when all of the conjuncts in its *cond* clause are satisfied. When one of these conjuncts is of the form **match** e, and when e matches a subexpression q of P, then an action of the form **rewrite**(w) is performed by replacing q with w.

Within the rules shown below, we only annotate the newly created SQ2$^+$expressions. The type information of other subexpressions are assumed unchanged. In addition to the first rewriting rule below, we only supply the rules for fixed point iteration, set comprehension and set unions. The rules for other SQ2$^+$constructs are either similar or trivial.

```
begin /* the rules */
/* initialization */
     if match e then
          enter(typev, [e, t]), /* enter the tuple [e, t] to the relation typev */
          where t = newvar. /* generate a new type variable t */
/* set union */
     if match x ∪ y, typev(x, t1), typev(y, t2), typev(x ∪ y, t3) then
          enter(subtype, [t1, set(m)]), enter(subtype, [t2, set(m)]),
          enter(typev, [x ∪ y, set(m)]), where m = newvar.
/* set comprehension */
     if match {e1: x ∈ e2 | e3}, typev(e1, t1), typev(e2, t2), typev(e3, t3),
               typev(x, t4), typev({e1: x ∈ e2 | e3}, t) then
          enter(subtype, [t2, set(t4)]), enter(subtype, [t3, bool]),
          enter(typev, [{e1: x ∈ e2 | e3}, set(t1)]);
/* fixed point iterations */
     if match x_e1 = e2; e3, typev(x, t0), typev(e1, t1), typev(e2, t2)
               typev(e3, t3 ), typev(x_e1 = e2; e3, t4) then
          enter(typev, [x, t1]), enter(typev, [x, t2]),
          enter(typev, [x_e1 = e2; e3, t3])
end;
transcript typing;
declare: insertions of type need unification;
external: typev, subtype; /* transcript subtyping must be executed first */
begin
     if typev(e, t) then
          enter(type, [e, t]).
/* unify s with t */
     if type(e, t), subtype(t, s) then
          enter(type, [e, s]);
end;
```

We can show that if *type* is computed successfully by the transcripts above, then relation *subtype* of the first transcript is a type containment set of a principal simple subtyping of the input SQ2$^+$program e. Let C be the type containment set represented by the relation *subtype*. Let G(C) be the graph defined in Section 5. If G(C) has a cycle, then the built-in unification procedure will report a type error when computing the relation *type*. The declaration that *subtype* needs unification guarantees the unique coercion condition. It is also not so difficult to show by induction that each time when a tuple [x, y] is entered into relation *subtype*, x is always a variable, and y is either a type constant or a nonnested structure (e.g., set(m), where m is a variable).

Our SQ2$^+$compiler uses a set theoretic fixed point transformation [CP89] and a naive form of finite differencing [PK82] to transform SQ2$^+$programs into SETM code (see Section 2), which corresponds closely to the operational semantics given in the kernel language for SQ2$^+$. The base information inferred for the SQ2$^+$queries is extended to base information for the compiled low-level SETM code. For simplicity's sake we shall demonstrate this process informally below for naive SETM code generation.

compiler has three phases. First, type inference finds the principal simple subtyping $C, A \vdash e : \tau$ for a given SQ2$^+$program e. Next, program e is transformed into SETM code e' in which variables in common to both e and e' have types defined by A, and new variables in e' have types derived from A in a straightforward way; i.e., $C, A \cup A' \vdash e' : \tau$, where A' is a type assignment for all variables of e' not occurring in e. Finally, the SETM code is transformed into an efficient C program by the real-time simulation technique described in Section 2.

In this section we will sketch the RAPTS implementation (except that Ada will be the target language instead of C), and provide links with the subtyping theory crucial to the real-time-simulation. Within RAPTS subtype inference is implemented using an inductive rule system similar to the natural semantics of MENTOR/TYPOL [DGHKL84, Kah87]. However, while MENTOR/TYPOL uses a logic programming paradigm with a PROLOG back-end, RAPTS uses an expert system technology based on a home-grown variant of the RETE algorithm [For82] supported by highly efficient bottom-up linear pattern matching [CPT90].

Fragments of two rule groups, called *transcripts*, that infer principal simple subtypings for SQ2$^+$programs in RAPTS are shown below. The first transcript computes the type containment set *subtype* for an SQ2$^+$program e; i.e. if $t \leq \tau$, then $subtype(t, \tau)$. This transcript also computes the type assignment *typev* for e, where assertion $x : \tau$ is represented by $typev(x, \tau)$.

The second transcript uses *typev* and *subtype* to compute another type assignment *type* for the principal typing of an SQ2$^+$program e. The computation fails with an occurs check only when e is not typable. When the computation succeeds, we can get the principal typing of e by restricting *type* to the free variables of e.

Each rule in the transcripts below is of the form

if *cond* **then** *action* **where** *binding*

where *cond* is a set of conjuncts separated by commas, and *action* is a collection of actions to be performed if all of the conjuncts in *cond* are true. These rules are applied to a given program P. When one of the conjuncts is of the form **match** e, then the rule will only be used when pattern e matches a subexpression q of P. Matching binds pattern variables of e to subtrees of q. The other conjuncts may result in further bindings. The *binding* clause contains subclauses of the form x = **newvar**, which binds pattern variable x to a newly generated identifier. If all of the conditions in *cond* hold, then the resulting substitution is used to instantiate the actions. Actions of the form enter($reln$, [x1,..,xk]) enter k-tuple [x1,...,xk] into relation $reln$. However, if the transcript declares that $reln$ needs unification, then each time a tuple [x1,...,xk] is entered into $reln$, the system will check whether there exists a tuple [x1', ...,xk'] in $reln$ such that x1' = x1. If not, [x1,...,xk] is added. Otherwise, x2,...,xk will be unified with x2',...,xk'. If the unification succeeds, a substitution will be performed using the most general unifier of x2',...,xk' and x2,...,xk. Otherwise, a type error is reported, and the input SQ2$^+$expression is not subtypable.

transcript subtyping;
declare: insertions of *subtype* need unification;
insertions of *typev* need unification;

$$\frac{\begin{array}{l} I \leq \mathrm{set}(A) \text{ (type containment set)} \\ X : \mathrm{set}(A) \\ F : \mathrm{set}(I, A) \text{ (type assignment)} \end{array}}{\begin{array}{l} D(S : \mathrm{set}(A)) : \mathrm{set}(A) = \\ \{a : (Y : I, a : A) \in F \mid [I \leq \mathrm{set}(A)]Y \subseteq S\}; \\ X^+ : \mathrm{set}(A) = X \cup D(X^+); \\ X^+ : \mathrm{set}(A) \end{array}}$$

The initial constraints for a type valuation T are

$$\begin{array}{rcl} T[I] & \subseteq & T[\mathrm{set}(A)] \\ \{\rho(X)\} & \subseteq & T[\mathrm{set}(A)] \\ \{\rho(F)\} & \subseteq & T[\mathrm{set}(I, A)] \end{array}$$

Because of $S \subseteq T[\mathrm{set}(\tau)] \Leftrightarrow (\bigcup S) \subset T[\tau]$ this is equivalent to

$$\begin{array}{rcl} \bigcup T[I] & \subseteq & T[A] \\ \rho(X) & \subseteq & T[A] \\ \rho(F) & \subseteq & T[(I, A)] \end{array}$$

Using $S' \subseteq T[(\tau', \tau'')] \Leftrightarrow (\mathbf{domain}\ S \subseteq T[\tau'] \wedge \mathbf{range}\ S \subseteq T[\tau''])$ this is, in turn, equivalent to

$$\begin{array}{rcl} \bigcup T[I] & \subseteq & T[A] \\ \rho(X) & \subseteq & T[A] \\ \mathbf{domain}\ \rho(F) & \subseteq & T[I] \\ \mathbf{range}\ \rho(F) & \subseteq & T[A] \end{array}$$

Finally, applying $(S \subseteq T[t] \wedge S' \subseteq T[t]) \Leftrightarrow S \cup S' \subseteq T[t]$ twice we arrive at

$$\begin{array}{rcl} (\bigcup T[I]) \cup \rho(X) \cup \mathbf{range}\ \rho(F) & \subseteq & T[A] \\ \mathbf{domain}\ \rho(F) & \subseteq & T[I] \end{array}$$

Note that $G(C)$ contains an edge from I to A for this example. So we calculate the most specific type valuation by first defining $T_0[I]$ and then $T_0[A]$.

$$\begin{array}{rcl} T_0[I] & = & \mathbf{domain}\ \rho(F) \\ T_0[A] & = & (\bigcup T_0[I]) \cup \rho(X) \cup \mathbf{range}\ \rho(F) \\ & = & (\bigcup \mathbf{domain}\ \rho(F)) \cup \rho(X) \cup \mathbf{range}\ \rho(F) \end{array}$$

For given ρ we say $T_0[I]$ is the base for I and $T_0[A]$ is the base for A. ■

7 Transforming SQ2$^+$queries into set machine code

Our subtyping theory provides the formal underpinnings for an SQ2$^+$compiler implemented by Cai and Paige in the RAPTS transformational programming system. The

This theorem says that given a subtyping $C, A \vdash e : \tau$ for an SQ2$^+$query e, we can compute unique 'minimal' bases, which are expressible as SQ2$^+$queries with the same free variables as e. Informally, this is justified by two claims. First, because type assignment A only has assumptions for free variables, we cannot use our type and subtype inference rules of Appendices B and C to derive type assertions $e : t$, where t is a type variable and SQ2$^+$expression e computes a *created* value (meaning that e does not retrieve an input value). For example, we cannot derive $S \cap T : t$ or $x^2 : t$, where t is a type variable. Consequently, $T[t]$ is defined in terms of input values. Second, type assignment A and type containment set C are associated with consistent set theoretic constraints that can be satisfied constructively with a unique minimal interpretation $T[t]$ as a finite set.

This theorem can be proved as follows. Let ρ be legal. By definition, T is a type valuation for ρ w.r.t. C, A if and only if

1. for every $x : \tau$ in A we have $\{\rho(x)\} \subseteq T[\tau]$ and
2. for every $\tau \leq \tau'$ in C we have $T[\tau] \subseteq T[\tau']$.

Note that we have the equivalences

1. $S \subseteq T[\mathrm{set}(\tau)] \Leftrightarrow (\bigcup S) \subseteq T[\tau]$,
2. $S' \subseteq T[(\tau', \tau'')] \Leftrightarrow (\mathbf{domain}\ S \subseteq T[\tau'] \wedge \mathbf{range}\ S \subseteq T[\tau''])$ and
3. $(S \subseteq T[t] \wedge S' \subseteq T[t]) \Leftrightarrow S \cup S' \subseteq T[t]$.

where $\bigcup S$ denotes the union of all element sets of S, $\mathbf{domain}\ S'$ is the domain of the set of pairs S', and $\mathbf{range}\ S'$ is the range of S'. Using them we can rewrite the initial constraints into an equivalent set of constraints such that all constraints are of the form $S \leq T[t]$ for some type variable t and there is at most one constraint of the form $S \leq T[t]$ for every type variable t. We call S the *lower bound* for t. If we draw an edge from every type variable in S to t for every $S \leq T[t]$ in the final constraint set it is easy to see that the resulting graph on type variables is just $G(C)$ as defined in Section 5, which is guaranteed to be acyclic. Thus the constraint set has a unique minimal solution T_0. In particular, we define T_0 as follows: For every source t in $G(C)$, we define $T_0[t] = \{\}$ if t has no lower bound, and $T_0[t] = S$ if S is the lower bound for t. For all other type variables t' in $G(C)$ we define, in topological order, $T_0[t'] = S[T_0/T]$ where S is the lower bound for t' and $S[T_0/T]$ says that the formal occurrence of T is to be interpreted by T_0. Note that $T_0[t]$ is finite and can be expressed as an SQ2$^+$query in terms of the (program) variables occurring in A since $\bigcup, \mathbf{domain}$ and $\mathbf{range}$ are definable in SQ2$^+$.

This proof sketch can be illustrated using the attribute closure example once again.

> **Example:** (Base calculation for attribute closure query) We shall present the base interpretation for the following (simple) subtyping for the attribute closure query.

can be extended in a canonical fashion to a mapping from type expressions to subsets of U by defining $T[\tau] = \{v : (\exists e)\, e \to v : \tau\}$. In particular, we have

$$\begin{array}{lcl} T[\mathbf{integer}] & = & \{\ldots, -1, 0, 1, 2, \ldots\} \\ T[\mathbf{bool}] & = & \{\mathbf{false}, \mathbf{true}\} \\ T[(\tau, \tau')] & = & T[\tau] \times T[\tau'] \\ T[\mathrm{set}(\tau)] & = & \{S : S \subseteq T[\tau] \mid |S| < \infty\} \end{array}$$

An *environment* ρ is a set of value assumptions of the form $x \to v : \tau$ for the free variables of an SQ2$^+$query e. The values bound to the free variables represent a particular input for e. We write $\rho(x)$ for the value bound to x in ρ.

We say ρ, T *satisfy* C, A if for all $x : \tau$ in A there is $x \to v : \tau'$ in ρ such that $v \in T[\tau]$, and for all $\tau \leq \tau'$ in C we have $T[\tau] \subseteq T[\tau']$. Notice that every pair C, A is satisfiable if C is a type containment set.

We write $C, A \models e : \tau$ if for all ρ, T satisfying C, A we have that $v \in T[\tau]$ whenever $e \to v : \tau'$ is derivable from ρ in the operational semantics of SQ2$^+$(see Appendix A). The soundness of subtypings w.r.t. the operational semantics of SQ2$^+$is easily formulated now.

Theorem 4 *(Semantic soundness)*
If $C, A \vdash e : \tau$ *then* $C, A \models e : \tau$.

6 A base interpretation of subtypings

In this section we show that the type variables occurring in a subtyping $C, A \vdash e : \tau$ can be interpreted in a unique fashion as finite sets of values constructed from the input values for the SQ2$^+$query e. These sets, constructed exclusively from the inputs (i.e., values of global variables) of a query, will serve as our bases, and exactly those variables in e whose types have an occurrence of a particular type variable t will be represented by aggregation around the base computed for t. Principal simple subtypings are the "best" subtypings for base interpretation since they provide the greatest "degree of freedom" for interpreting type variables without admitting costly compound dereferencing. Note, however, that the results of this section hold for *any* subtyping, not just simple subtypings.

For a well-typed SQ2$^+$query e the type containment set C and the type assignment A of a subtyping $C, A \vdash e : \tau$ for e can be interpreted as input requirements that must be satisfied for an input environment: An environment ρ is *legal* for C, A if there exists a type valuation T such that ρ, T satisfies C, A; we say that such a T is a *type valuation for* ρ *(w.r.t.* C, A*)*. A type valuation T is *at least as specific as* T', $T \leq T'$, if $T[t] \subseteq T'[t]$ for every type variable t.

Theorem 5 *Let* C *be a type containment set and* A *be a type assignment. For every legal environment* ρ *there is a (unique) most specific type valuation* T *such that* ρ, T *satisfies* C, A*. Furthermore,* T *maps every type variable to a finite set and for every type variable occurring in* C *there is an SQ2*$^+$ *query* e_t *such that* $\rho \vdash e_t \to T[t] : set(t)$ *in the operational semantics of SQ2*$^+$ *(see Appendix A).*

Just as there are principal typings for well-typed SQ2$^+$queries there is a corresponding notion for subtypings that has, in an intuitive sense, a maximum "degree of freedom" for instantiating type variables if we restrict ourselves to subtyping derivations that are especially suited to a base interpretation. We say a subtyping derivation is *simple* if

1. the compatibility rules (PAIR-COMP) and (SET-COMP) are not used, and
2. the (COERCE) rule is only invoked on those subexpressions that are used in an assumption for a type inference rule that *requires* a type expression with a particular type constructor or constant type for that assumption.

A subtyping is *simple* if it has a simple derivation. These conditions may sound obscure at first. But the first condition prevents the use of "induced" type containments $\mathrm{set}(t) \leq \mathrm{set}(\mathrm{set}(t'))$ in coercions; such a coercion would correspond to an exhaustive elementwise dereferencing of a whole set of references. The second condition makes sure that coercions are only invoked where data are "used", not where they are generated. For example, to type expression $\{e : x \in S \mid P\}$ we must derive $S : \mathrm{set}(\tau)$ for some type expression τ and $P : \mathbf{bool}$, but there is no requirement that e have a type with a particular form. Consequently, the second requirement says that the (COERCE) rule may be invoked on subexpressions S and P, but not on e.[7]

Let C be a type containment set. We write $C \vdash C'$ for an arbitrary collection C' of type containments (not necessarily a type containment set) if $C \vdash \tau \leq \tau'$ for every $\tau \leq \tau'$ in C' using the subtype inference rules of Appendix C. We say an SQ2$^+$query e has a *principal simple subtyping* $C, A \vdash e : \tau$ if it is a simple subtyping and for any simple subtyping $C', A' \vdash e : \tau'$ there is a substitution S such that $A' = S(A), \tau' = S(\tau)$ and $C' \vdash S(C)$. If they exist, principal simple subtypings are unique modulo renaming of type variables.

Theorem 3 *(Principal simple subtyping property)*
Every well-typed SQ2$^+$query has a principal simple subtyping.

The proof of this theorem is by reduction to type constraint systems[8] and is outside the scope of this article. Later, we will see that a derivation of the principal simple subtyping of an SQ2$^+$query is used by our SQ2$^+$compiler to obtain a 'best' implementation in the sense that the real-time simulation conditions described earlier are satisfied.

Finally, we show that subtypings (whether simple or not) and thus also typings are *sound*: they give valid information about the semantics of SQ2$^+$. To express this we first introduce some basic notions. Let U be a universe containing all "defined" values of SQ2$^+$queries. A *type valuation* T is a mapping from type variables to subsets of U. T

[7] These two conditions can be encoded by dropping all the type containment inference rules of Appendix C but rule (REFL) and "building" applications of the (COERCE) rule into the type inference system of Appendix B. This is done by systematically replacing every assumption of the form $e : \mathrm{set}(\tau)$ or $e : (\tau', \tau'')$ by $e : \sigma, \sigma \leq \mathrm{set}(\tau)$, respectively $e : \sigma, \sigma \leq (\tau', \tau'')$, where σ is a new meta-variable.

[8] See [Wan87, FM88, Hen88] for a sample of reductions of type inference to type constraint systems and [Hen91b, Hen91a] for reductions very similar to the one necessary to prove this theorem.

operations while navigating on a path (that does not pass through the same base twice) uniquely associated with B. It also ensures, for example, that all of the elements of a set have the same depth of nesting.

An SQ2$^+$query e is *subtypable* if there is a type containment set C, a type assignment A for e, and a type expression τ such that $e : \tau$ is derivable under C, A using only the inference rules of Appendices B and C. In this case we call $C, A \vdash e : \tau$ a *subtyping* for e.

Since every simple typing is also a subtyping with an empty type containment set, every simply typable SQ2$^+$query is also subtypable. Conversely, the properties of a type containment set guarantee that every type containment set C has a unifier; i.e., a substitution S such that $S(\tau) = S(\tau')$ for every $\tau \leq \tau'$ in C [Ede85]. So every subtyping derivation results in a simple typing derivation by applying S to the whole derivation. This proves the following theorem.

Theorem 2 *(Conservative extension)*
An SQ2$^+$query e is typable if and only if it is subtypable.

We represent a subtype derivation for an SQ2$^+$query e by annotating the bound variables of e with type expressions and indicating type containments $\tau \leq \tau'$ (used when applying the COERCE rule of Appendix C) explicitly in square brackets in a subtyping $C, A \vdash e : \tau$ for e. This identifies a subtype derivation for e modulo proofs of type containments. Later we will show how our SQ2$^+$compiler is guided by derivations, so that different derivations can lead to different implementations.

The main value of subtypings should thus be seen as a characterization of typability while providing more detailed information for well-typed SQ2$^+$queries than (simple) typings. This additional information can be illustrated by the following derivation of attribute closure.

Example: (A subtyping derivation for attribute closure) The attribute closure specification of Section 3 admits the following subtyping derivation.

$$\frac{\begin{array}{l} t_X \leq \mathrm{set}(A) \\ t_F \leq \mathrm{set}(I, A) \\ I \leq \mathrm{set}(A) \text{ (type containment set)} \\ X : \mathrm{set}(A) \\ F : \mathrm{set}(I, A) \text{ (type assignment)} \end{array}}{\begin{array}{l} D(S : \mathrm{set}(A)) : \mathrm{set}(A) = \\ \{a : (Y : I, a : A) \in [t_F \leq \mathrm{set}(I, A)]F \mid [I \leq \mathrm{set}(A)]Y \subseteq S\}; \\ X^+ : \mathrm{set}(A) = [t_X \leq \mathrm{set}(A)]X \cup D(X^+); \\ X^+ : \mathrm{set}(A) \end{array}}$$

Note that this subtyping derivation is different from the simple typing derivation of Section 4 in that it has additional type containment assumptions $t_X \leq \mathrm{set}(A), t_F \leq \mathrm{set}(I, A), I \leq \mathrm{set}(A)$, which are used once each in the derivation. The interpretations of the type variables I and A are crucial in the construction of bases for attribute closure (see Sections 6 and 7). ■

preserving, then we would need an extended type system with a Damas/Milner [Mil78, DM82] style let-polymorphism, which has a DEXPTIME-complete type inference problem [Mai90, KTU90]. Note that unresolved overloaded function symbols in other languages may destroy the principal typing property.

5 A subtyping system for SQ2$^+$

Type inference can be employed to collect information about SQ2$^+$queries. The simple type inference system for SQ2$^+$of the previous section only provides information about *equality* of types. For base analysis we are interested in *containments* between types since, as we shall see later, type variables can be interpreted as bases, and one base may be related to another base; e.g., the elements in a base B may be sets of elements of another base B'. This relationship between two bases is captured by the *type containment* $t \leq \mathrm{set}(t')$, where t and t' are type variables and $\leq$ is interpreted as containment between the bases denoting the types t and $\mathrm{set}(t')$. If there is exactly one type containment for t on the left-hand side, then t can be implemented as a set of pointers ("addresses") to memory cells containing elements of type $\mathrm{set}(t')$. Pointer dereferencing is then the operational analogue to invoking the conversion rule for a value of type t to get a value of type $\mathrm{set}(t')$.

We refine the simple type inference system for SQ2$^+$to a *subtype inference system* (c.f. [Mit84, FM88]) by adding propositional formulas of the form $\tau \leq \tau'$ and the rule schemes presented in Appendix C. The rules specify that type containments are reflexive, transitive and closed with respect to type constructors set(.) and (.,.). The rule (COERCE) connects the type containments to SQ2$^+$queries: Any SQ2$^+$query of type τ is also of type τ' if $\tau \leq \tau'$ is derivable.

For an arbitrary set C of type containments we define the directed graph $G(C)$ on the type variables occurring in C such that there is an edge from t to t' if and only if there is a constraint $t \leq \tau$ in C and t' occurs in τ.

A set C of assumptions of the form $\tau \leq \tau'$ is a *type containment set* if

1. (type variables on the left of containments) for all $\tau \leq \tau'$ in C the left-hand side, τ, is a type variable;

2. (unique coercions) for $\tau \leq \tau', \tau \leq \tau''$ in C it is the case that $\tau' = \tau''$;

3. (acyclic coercions) $G(C)$ is acyclic.

These three properties restrict the data structures formed from bases in a natural way. (In the next section we will show how each type variable in a subtyping can be uniquely interpreted as a base.) The first property states that a value of type variable τ (essentially a pointer to an element of a base) can be coerced by a single application of dereferencing into a value of type τ'. The second property states that the type of the value obtained by this dereferencing is unique. This results in unique navigation from one base to another. The third property ensures that the set of bases related by dereferencing forms a partial ordering. Among other things, this means that the unique value of simple type for each element of a given base B can be obtained by a performing a finite number of dereferencing

$$\text{(SETCOMP)}\quad \frac{\begin{array}{l} M : \mathrm{set}(\tau) \\ x : \tau \vdash P : \mathbf{bool} \\ x : \tau \vdash e : \tau' \end{array}}{\{e : x \in M \mid P\} : \mathrm{set}(\tau')}$$

which is derived from the typing rules for set iteration, the empty set and element insertion since $\{e : x \in M \mid P\} : \mathrm{set}(\tau')$ is defined by $\{\}, e : x \in M \mid P$ in the kernel language.

In general there may be several derivations of a typing for an SQ2$^+$query e. A particular derivation can be uniquely represented by *annotating* the bound variables of e with type expressions in a typing $A \vdash e : \tau$.

> **Example:** (A typing derivation for attribute closure) Consider the attribute closure specification of Section 3:
>
> $D(S) = \{a : (Y, a) \in F \mid Y \subseteq S\};$
> $X^+ = X \cup D(X^+);$
> X^+
>
> This is a well-typed SQ2$^+$query, and a particular typing derivation is given by the annotated typing
>
> $$\frac{\begin{array}{l} X : \mathrm{set}(A) \\ F : \mathrm{set}(\mathrm{set}(A), A) \text{ (type assignment)} \end{array}}{\begin{array}{l} D(S : \mathrm{set}(A)) : \mathrm{set}(A) = \{a : (Y : \mathrm{set}(A), a : A) \in F \mid Y \subseteq S\}; \\ X^+ : \mathrm{set}(A) = X \cup D(X^+); \\ X^+ : \mathrm{set}(A) \end{array}}$$
>
> In this derivation the attribute closure query has type $\mathrm{set}(A)$ for type assignment $\{X : \mathrm{set}(A), F : \mathrm{set}(\mathrm{set}(A), A)\}$. Since A is a free type variable this holds for *any* type expression substituted for A. ■

An SQ2$^+$query e has a *principal typing* $A \vdash e : \tau$ if for any typing $A' \vdash e : \tau'$ for e there is a substitution S such that $S(A) = A'$ and $S(\tau) = \tau'$. If they exist, principal typings are unique modulo renaming of type variables. The derivation represented in the above example is for the principal typing of the attribute closure query.

The following theorem is well-known in simple parametric type systems for functional languages [Cur69, Hin69, DM82].

Theorem 1 *(Principal typing property)*
Every well-typed SQ2$^+$query has a principal typing.

Theorem 1 follows from the facts that the set of typing derivations for SQ2$^+$query e can be characterized as the set of unifiers of a pair of type expressions derived from e and that every unifiable pair of type expressions has a most general unifier. All the constants of SQ2$^+$have a principal typing. Consequently, principal typings can be computed efficiently by reduction to unification. Of course, eliminating auxiliary functions by unfolding can result in an exponential expansion in the length of the program text. Alternatively, if unfolding auxiliary functions in SQ2$^+$expressions is assumed to be meaning (and type)

4 A type inference system for SQ2$^+$

SQ2$^+$as presented in Section 3 and Appendix A is a dynamically typed language, because alternative results in conditional expressions can have different types (cf. rules COND-I and COND-II of Appendix A). We shall use type inference primarily as a tool for program analysis and program transformation. In this sense we can view a static type inference system as a specialized program logic, even for dynamically typed languages. Programs that cannot be statically typed are not subjected to transformations that need such type information to be applicable. Since such programs are of no interest to us, we shall restrict ourselves to a statically typed subset of SQ2$^+$queries. Queries in this subset are said to be *well-typed*. Well-typed queries have the additional benefit that they are guaranteed to have no type errors during evaluation.

In this section we present a first-order parametric type inference system for SQ2$^+$, and in Section 5 we generalize it to a subtype inference system that includes type containments. We shall see in Section 6 how subtype inference can be used to find bases.

Our type system is a variant of the Curry/Hindley type discipline for the λ-calculus (resp. combinatory logic) [Cur69, Hin69]. It is parametric since it permits the presence of type variables, but it is not polymorphic, since auxiliary functions cannot have a polymorphic type. We shall see in Section 6 that the type variables are of particular significance in base analysis.

The full type inference system for the kernel of SQ2$^+$can be found in Appendix B. A set A of assumptions of the form $x : \tau$, one for each free variable x in e, is a *type assignment for* e. We shall view A as a map on (program) variables and denote the type in the typing assumption for x with $A(x)$. An SQ2$^+$query e is *(simply) typable* (or *well-typed*) if there is a type expression τ and a type assignment A for e such that $A \vdash e : \tau$; i.e., $e : \tau$ is derivable under A using only the inference rules of Appendix B. In this case we say $A \vdash e : \tau$ is a *typing* for e.[6]

A basic language principle we follow is that exactly those SQ2$^+$queries that can have a type represent "first-class" values. Since SQ2$^+$is a first-order language, (auxiliary) functions are not values, and consequently they cannot have a type, yet function *applications* do have types. Consequently we distinguish between atomic formulas of the form $e : \tau$ and $f ::< \tau, \tau' >$, the latter of which may be used as (hypothetical) assumptions in typing derivations. Having no function types guarantees that an SQ2$^+$query $f(x) = \ldots; f$ is *not* well-typed (and has no value) since there cannot be a typing for it. However, $f(x) = \ldots; f(c)$ may be well-typed. Even if e is typable this does not imply that e has a value since its evaluation may fail to terminate (in which case it has no value).

The type inference system is systematically derived from the operational semantics of Appendix A with "strict" (abstract) interpretation of conditionals and function binding. Consequently it is not very difficult to show that the typing rules are semantically sound. We defer a formal statement of soundness until Section 5 when type containments are added to our type inference system.

For set comprehension we obtain the typing rule (scheme)

[6]Generally, we require that the set of "active" typing assumptions at any point in a proof forms a type assignment.

a set of vertices V it computes the set of vertices one edge away from V. To evaluate the full query, X^* is initialized to the empty set. Since $N(\{\}) = \{\}$, the value of X^* after the first iteration is X, and then, in successive iterations, the neighbors of the "current" set of reachable vertices are added to X^* until no more vertices are added in this way. The final value of X^* is the answer. ■

The much more efficient low-level SETM code for reachability, shown at the end of the previous section, can be derived from this query by fixed point and finite differencing transformations as implemented in RAPTS [CP89].

Example: (Powerset) To underscore that SQ2$^+$can also manipulate nested sets, unlike relational calculus, here is a specification of the powerset of a given set S.

$\mathcal{P} = \{\{\}\} \cup \{\{x\} \cup T : x \in S, T \in \mathcal{P}\};$
$\mathcal{P}$

$\mathcal{P}$ is initialized to $\{\}$ and becomes $\{\{\}\}$ after one iteration. After k iterations it contains all subsets of S with at most k elements. Hence convergence takes $|S|$ iterations. ■

The following example specifies the attribute closure of a set of functional dependencies for a set of attributes X. It is noteworthy since it contains nested sets and admits a basing that can be found using type inference *with* type containments, but not without them (see Sections 4 and 5).

Example: (Attribute closure) A functional dependency is a pair consisting of a set of attributes Y and an attribute a. In database theory such a pair expresses that the (values of the) attributes in Y directly "determine" (the value of) attribute a. Given a set of such functional dependencies F and a set of attributes X the attribute closure of X under F is the set X^+ of attributes whose values are determined by the values in X (under F); i.e., they are in the reflexive-transitive closure of X under F. It can be specified as follows in SQ2$^+$.

$D(S) = \{a : (Y, a) \in F \mid Y \subseteq S\};$
$X^+ = X \cup D(X^+);$
X^+

$D(S)$ returns the attributes whose values are determined *directly* (in one inference step) by S. As in the previous examples, X^+ is initialized to the empty set. Since $D(\{\}) = \{\}$, X^+ is assigned X after the first iteration. At each iteration newly determined attributes are added to X^+ until this process converges. The final value of X^+ is then returned. ■

(by value) in e_2. If x denotes a pair we may also use the pattern matching notation $f(x_1, x_2) = e_1; e_2$. A fixed point definition $x_f = e_1; e_2$ is evaluated by *fixed point iteration*: first x is initialized with the value of f; next the assignment $x := e_1$ is executed repeatedly until two successive values of x are equal. This value is then used as the value of x to evaluate e_2. If the subscript f in $x_f = e_1; e_2$ is missing, we assume a type-specific default initialization for x: $\{\}$ for sets, 0 for integers, **false** for Booleans, and a pair (d_1, d_2) of such default values for pairs.

A complete formal structural operational semantics [Set89, chapter 11] for a kernel of SQ2$^+$is presented in Appendix A using a natural deduction style [GLT89, chapter 2] inference system. We use assertions of the form $e \rightarrow v : \tau$, which express that SQ2$^+$query e has value v of type τ. Types are expressions generated by the production $\tau ::= \mathbf{integer} \mid \mathbf{bool} \mid (\tau_1, \tau_2) \mid \mathrm{set}(\tau) \mid t_1, t_2, \ldots$, where $t_1, t_2, \ldots$ are type variables. The values contain the truth values **false**, **true**, the integers $\ldots, -1, 0, 1 \ldots$, as well as pairs (v_1, v_2) and finite sets $\{v_1, \ldots, v_k\}$ of such values. The kernel contains a general set iterator construct from which set comprehension and other operations may be defined. Informally, if M is a set and P is a predicate, then the result r of a set iteration $e' \oplus e : x \in M \mid P$ is computed by initializing r to e' and executing $r := r \oplus e$ for every x in M that satisfies P (e and P may contain free occurrences of x).

SQ2$^+$is nondeterministic since e may have several distinct values as well as terminating and nonterminating computations for the same input. We shall not concern ourselves with the potentially problematic interaction of nondeterminism and fixed point iteration since it is irrelevant for our purposes. Suffice it to say that, in general, there may be both terminating and nonterminating computations for fixed point iteration. However, if the function denoted by e_1 in $x = e_1; e_2$ is monotonic with respect to values for x relative to some partial order $\leq$ — such as containment, $\subseteq$, for sets — and if the initial value for x is minimum w.r.t. $\leq$ then all possible computations of x have the same observable effect: either they fail to terminate or they return a unique value for x, the least fixed point of e_1 (least w.r.t. $\leq$). This property has been exploited and refined in transformations of least/greatest fixed point specifications to low-level set-theoretic code [PH87, CP89]. Since those transformations are not the subject of this paper we shall not dwell on them, however.

3.2 Examples of SQ2$^+$queries

To get an idea of the expressiveness and conciseness of SQ2$^+$we present some illustrative examples.

> **Example:** (Reachable vertices) Given source vertices X and binary edge relation E, the set X^* of vertices reachable from X following paths in E is given by
>
> $N(V) = \{y : (x, y) \in E \mid x \in V\}$;
> $X^* = X \cup N(X^*)$;
> X^*
>
> Note that N is an auxiliary function introduced only for the sake of clarity. (All such auxiliary functions can, in principle, be eliminated by unfolding.) For

$$
\begin{array}{rcl}
e & ::= & x \\
& | & c \\
& | & e_1 \oplus e_2 \\
& | & (e_1, e_2) \\
& | & fe \\
& | & \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \\
& | & fx = e_1; e_2 \\
& | & x_{e_1} = e_2; e_3 \\
& | & \{e_1 : x_1 \in e_2, \ldots \mid e_3\} \\
x & ::= & \textit{<variable>} \\
c & ::= & \textit{<constant>} \\
f & ::= & \textit{<function symbol>} \\
\oplus & ::= & \textit{<binary operator>}
\end{array}
$$

Figure 1: Abstract syntax of SQ2$^+$

In the following pages we will show how analysis for universal sets, inclusion and membership tests, and data structure choice can be achieved by *base analysis* of a language of very high level, which draws on fundamental concepts of type theoretic calculi for subtyping and inheritance. We believe subtype analysis is a promising novel approach to global data structure optimization.

3 SQ2$^+$: A set query language

The language SQ2$^+$ (set queries with fixed points) can be viewed as an extension of relational calculus with first-class sets, arithmetic, nonrecursive function definitions and fixed point definitions. In this sense it is similar to database query languages for nonflat relational databases, but it can also be understood as extending a declarative variant of the (imperative) programming language SETL. A complete description of the *kernel* syntax of SQ2$^+$ is given as part of the type inference system for SQ2$^+$ in Appendix B. It is a variant of an earlier language called SQ+ [PH87, CP89], and can be shown to be Turing-complete like SQ+ (cf. Theorem 20 in [CP89]). A partial description of SQ2$^+$ is given in Figure 1.

3.1 Informal semantics of SQ2$^+$

The free variables in an SQ2$^+$ query e are the inputs of e, and the result is the[5] value of e. The value of e is calculated as in SETL expressions. Expression $fx = e_1; e_2$ defines a nonrecursive auxiliary function f with parameter x and body e_1 that can be called

[5] We shall, somewhat carelessly, say "the" value of a query even though there may be several possible answers; i.e., SQ2$^+$ queries are nondeterministic in general.

are strongly based, initialization also takes unit time. However, in [Pai89] simulating re-initialization for sets S, whose elements are strongly based, using only list and pointer structures required $\Omega(|S|)$ time. Sometimes this cost can be charged to other operations. However, if we allow an array implementation, then a unit-time initialization (and re-initialization) can be achieved using the solution to exercise 2.12 of Aho, Hopcroft, and Ullman's book [AHU74] combined with Wiederhold's transposed column method [Wie83].

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 SETM code just below runs in worst case time $O(|e|)$ with respect to the complexities given in Table 1.

```
t1 := {}
(for x ∈ w) -- copy all elements of w to t1
    t1 with:= x
end
r := {} -- initialize r to the empty set
(while ∃a ∈ t1) -- while there is an element a in t1
    (for y ∈ e{a}) -- add all new neighbors of a to t1
        if y ∉ r and y ∉ t1 then
            t1 with:= y
        end
    end
    t1 less:= a -- delete a from t1
    r with:= a -- add a to r
end
```

If we let $v = w \cup \textbf{domain}\ e \cup \textbf{range}\ e$ be our base, then it is easy to prove the program invariants $t1, r \subseteq v$ and $a, x, y \in v$. Consequently, the SETM code can be simulated in real-time if the elements of **domain** e, r, and $t1$ 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, Ull82], 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.

Our data structures improve the SETL based representations in several ways. SETL lacked a linked list representation for sets with unbased elements, did not permit iteration over sets with strongly based elements, and required all sets with weakly based elements to be hashed (making element addition expensive). SETL also had more complicated memory management, because hash tables were required for bases and for sets with either unbased or weakly based elements.

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 loop
    ...
    Q less:= x -- delete x from set Q
    ...
end
```

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 the identifiers S and Q are the same, 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.

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. 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 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 S whose elements

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 (retrieval of **range** f)	$O(1)$
(**for** $x \in s$)	execute ... for each x in set s (searching cost only)	$O(\|s\|)$
...		
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 (**domain** f is accessed)	$O(1)$
$f\{x\}$	$\{y : [u, y] \in f \mid u = x\}$ (**domain** f is accessed)	$O(1)$
$f(x) :=$	indexed assignment to function (**domain** f is accessed)	$O(1)$
$f\{x\} :=$	indexed assignment to map (**domain** f is accessed)	$O(1)$
s **less** $:= x$	set element deletion	$O(1)$

Table 1: SETM Primitives

Next, we define a transformation T that uses familiar pointer structures to simulate SETM 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 SETM 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. Of course, transformation T is not always applicable, but when it is then $T(P)$ has the same asymptotic time and space complexity on a RAM as P in the abstract complexity model of SETM.

Our simulation does not include the cost of reading the input and *preconditioning* it by transforming it into the data structures used for real-time simulation of the set operations. As long as all of the input is read in a single batch operation, multiset discrimination [CP91] can be used to implement preconditioning in linear time, which shows that no "hidden" cost is introduced by way of "intelligent" — and costly — input operations.

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

It is useful to divide up the primitive SETM 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. SETM 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.

Real-time simulation of SETM 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 increases the SETM time complexity if the size of S depends on the size of the input. 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. Our approach is 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

are extended to the intermediate-level imperative code (similar to SETL) that results from the initial phase of SQ2$^+$compilation. Type variables occurring in SQ2$^+$subtypings are uniquely interpreted as finite universal sets, called *bases*. Bases are used for avoiding data replication and for creating aggregate data structures that implement logical associative access operations in the intermediate-level code by simpler cursor or pointer access on a RAM[AHU74]. The RAM implementation is obtained from the subtyping proof of the high level SQ2$^+$query together with a simple analysis of the intermediate-level code. We show that every SQ2$^+$query has a principal subtyping, and suggest that its proof leads to a 'best' RAM implementation. The RAM 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, (5) no type errors occurring at runtime, and (6) no unintended side-effects due to pointer aliasing.

Our method of choosing efficient representations for sets, maps, and other structured datatypes is part of a first attempt to implement the elementary algorithm design principles in [Pai89], which stem from the real-time simulation of an abstract sequential set machine on a uniform cost sequential random access machine (RAM) [AHU74]. We believe that further development of this approach could lead to the more ambitious data structure compiler envisioned by Tarjan.

The paper is organized as follows. Section 2 describes the set machine language SETM and data structure design principles that support the real-time simulation of SETM on a RAM. In Section 3 we describe our query language SQ2$^+$and give several illustrative examples. Its type model is explained in Section 4. In Section 5 the type model is refined to a model for inferring subtype relations without changing the class of typable SQ2$^+$expressions. In Section 6 we show how the type variables occurring in subtype relations for a set query can be interpreted as universal sets (bases), computed from the input in a unique fashion. Drawing on earlier work [PH87] we indicate in Section 7 how the base information gained can be maintained and refined during program transformation to low-level SETM code. We illustrate base analysis and data structure selection by transforming an SQ2$^+$specification of the relational attribute closure problem into SETM code, which is then translated into Ada. Section 8 reports some empirical results from implementing and comparing several attribute closure implementations with the one obtained from our data structure selection transformation.

2 Real Time Simulation of a Set Machine on a RAM

In this section we describe the data structure design method that our program optimization technique seeks to utilize. 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.

Following [Pai89], we first define a simple set machine (language) SETM and its associated abstract model of complexity. For each SETM primitive operation q, we define a worst case asymptotic time bound $O(f_q)$ for performing q on an idealized set machine.

occurring in type expressions. All remaining analysis to facilitate data structure selection is obtained by local rewriting that compiles these high level typed specifications into efficient RAM code.

Our method guarantees improved performance over the SETL default implementation by using highly efficient data structures without any hashing in order to simulate a set machine on a pointer machine in real time. We illustrate our data structure selection method by transforming a concise set-theoretic specification for attribute closure into low-level Ada code. We present some preliminary benchmark results that show substantial gains in efficiency over unoptimized code with hashed default implementations for sets and maps.

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]

and 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 quasi-algorithmic 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 that combines ideas from type theory and elementary algorithm and data structure design theory. We show a novel way in which global type information inferred in a very high level, declaration free, statically typed, Turing-complete, and functional first-order query language, called SQ2$^+$, can be compiled into efficient pointer structures that provide concrete and provably correct implementations in a lower-level language (e.g., one that is explicitly typed like Ada).

SQ2$^+$is distinguished by a rich collection of finite set theoretic expressions (as are found in a functional subset of SETL) and a fixed point operation. Subtypings (i.e. type containments) are inferred from SQ2$^+$queries in a first-order subtype theory. These subtypings

Type Analysis and Data Structure Selection

J. Cai[a,1], Ph. Facon[b], F. Henglein[c,2], R. Paige[a,3], E. Schonberg[a,4]

[a]New York University, Department of Computer Science, 715 Broadway, 7th floor, New York, NY 10003, USA

[b]CEDRIC-IIE (CNAM), Quartier les Passages, 18 Allee Jean Rostand, B.P. 77, 91002 Evry Cedex, France

[c]DIKU, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen Ø, Denmark

Abstract

Schwartz *et al.* described an optimization to implement built-in abstract types such as sets and maps with efficient data structures. Their 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 associative access by simpler cursor or pointer access. The SETL implementation used global analysis similar to classical dataflow for typings and for set inclusion and membership relationships to determine bases. However, the optimized data structures selected by this optmization did not include a primitive linked list or array, and all optimized data structures retained some degree of hashing. Hence, this heuristic approach did not guarantee a uniform improvement in performance over the use of default representations. The analysis was complicated by SETL's imperative style, weak typing, and low level control structures. The implemented optimizer was large (about 20,000 lines of SETL source code) and only partially operational.

In this paper we solve a modified form of Schwartz's data structure selection problem with a simpler top-down transformation that uses a first-order parametric subtyping theory for inferring typings and type containments in a high-level declarative set-theoretic programming language. Bases are determined from an interpretation of type variables

[1]The research of this author was partially supported by National Science Foundation grant CCR-9002428. Part of this work was done while the author was visiting the University of Wisconsin at Madison.

[2]The research of this author was partially supported by Office of Naval Research grants N00014-87-K-0461 and N00014-90-J-1110. Part of this work was done while the author was at New York University, USA, and at Utrecht University, The Netherlands.

[3]The research of this author was partially supported by Office of Naval Research Grant No. N00014-90-J-1890. Part of this work was done while the author was visiting the University of Wisconsin at Madison.

[4]The research of this author was partially supported by Office of Naval Research (DARPA) Grant No. N00014-90-J-1110.