Revision of paper in *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, Nice, France, June 1990, 218-226.

Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages

March, 1990

Henry G. Baker

Nimble Computer Corporation, Encino, CA 91436

1. Abstract

Type inference is the process by which an expression in an untyped computer language such as the lambda-calculus, Lisp, or a functional language can be assigned a static data type in order to improve the code generated by a compiler. *Storage use inference* is the process by which a program in a computer language can be statically analyzed to model its run-time behavior, particularly the containment and sharing relations among its run-time data structures. The information generated by storage use information can also be used to improve the code generated by a compiler, because knowledge of the containment and sharing relations of run-time data structures allows for methods of storage allocation and deallocation which are cheaper than garbage-collected heap storage and allows for the in-place updating of functional aggregates.

Type inference and storage use inference have traditionally been considered orthogonal processes, with separate traditions and literature. However, we show in this paper than this separation may be a mistake, because the best-known and best-understood of the type inferencing algorithms—Milner's unification method for ML—already generates valuable sharing and containment information which is then unfortunately discarded. We show that this sharing information is already generated by standard unification algorithms with no additional overhead during unification; however, there is some additional work necessary to extract this information. We have not yet precisely characterized the resolving power of this sharing and containment information, but we believe that it is similar to that generated by researchers using other techniques. However, our scheme seems to only work for *functional languages* like pure Lisp.

The unification of type and storage inferencing yields new insights into the meaning of "aggregate type", which should prove valuable in the design of future type systems.

This work was supported in part by the U.S. Department of Energy Contract #DE-AC03-88ER80663

2. Introduction

One of the major goals of modern higher-level language systems is the efficient allocation and deallocation of complex data structures. This goal has become increasingly important with the rise in popularity of *object-oriented, functional* and *persistent* programming paradigms. Typical techniques for implementing object-oriented programming paradigms involve the use of functions in the form of functional *closures* to simulate, and hide the details of, traditional data structures. The storage for these objects must be efficiently managed by the language, because this storage is not explicitly visible to the programmer. Functional languages require in addition the efficient implementation of lazy evaluation and functional updating of aggregates. Incorporating persistence into a system adds complexity to the implementation, because many *ad hoc* techniques of storage management which are acceptable for internal storage such as simply abandoning an address space, are no longer acceptable for persistent languages. These developments are all pushing programming systems in roughly the same direction—the employment of a garbage-collected heap.

Since the invention of garbage-collected heaps with the first implementation of the Lisp language, there has been an enormous amount of progress in garbage-collection algorithms, whether using a form of tracing (mark/sweep or copying), or some form of reference counting. The state-of-the-art garbage-collected heap offers some sort of generational collection [Lieberman83,Moon84,Ungar84], which provides for efficiency and responsiveness which is refreshing to those of us from an earlier era. If the system requires *hard* real-time response instead of excellent *stochastic* response, then the heap will offer some sort of *incremental* collection technique [Baker78,Appel88].

Nevertheless, garbage collection—or more generally—storage allocation and deallocation—remains a bottleneck in most high-level programming languages. The U.S. Department of Defense refuses to require garbage collection in its standard language—Ada [Ada83]—because it fears that implementations incorporating garbage collection will not be capable of meeting mission-critical time constraints ("[garbage collection] is rather costly and cannot be used effectively for real time systems, since it may occur unpredictably at critical times" [Ichbiah79,p.6-4]).

After many years of research into storage management techniques, it is still true that a clever programmer can handle allocation and deallocation more efficiently than the general purpose mechanisms used in higher level languages. Many researchers have treated this fact as a clue to the further improvement of storage management performance. These researchers have concluded that the programmer can do better, because he can see the global picture of what is going on in the program, whereas the run-time system can see only incrementally what is going on [Schwartz75, Muchnick76, Barth77, Steele78, Muchnick81, Pleban81, Hudak85, Ruggieri87, Chase87, Ruggieri88, Hederman88, Inoue88, Horowitz89, Jones89].

Therefore, in order to further improve performance in storage management, there has been some research attempting to gather more static information in order to reduce the cost of allocation and deallocation at run-time, and in order to implement "in-place" updating of aggregates in functional programming languages [Schwartz75, Hudak85, Bloss89]. We call the process of statically extracting information from a program about the run-time behavior of its storage management *storage use inferencing*, through an analogy with the extraction of datatype information statically through *type inferencing*.

Whereas the general thrust of type inferencing is the determination of the set of run-time values which could inhabit storage cells—these types are usually *independent* of one another—the general thrust of storage use inferencing is the determination of the *relations* among the run-time values which could inhabit the storage cells. If there were no relationships among the values in different storage cells, then there would be no sharing and no containment between data structures, and hence no problems in allocation, deallocation or updating.

The most common method for representing storage use relationships for static analysis has involved some sort of *graph* structure, which is a finite structure representing the possibly infinite set of storage relations which could occur at run-time for a program [Muchnick81, Jones82, Ruggieri87, Chase87, Hederman88, Larus89]. The exact nature of the approximations used has been the subject of several papers. Some of these representations and algorithms are on the verge of becoming elegant, at least for functional languages [Chase87].

The other great tradition in static analysis for untyped languages has been *type inferencing*. Type inferencing is the process whereby proper *type declarations* are obtained by a mechanical process from a program which neglected to use them [Milner78, Suzuki81, Mitchell84, Wand84, Boehm86, Peyton-Jones87, Wand89, Baker90]. If we consider a legal program which has complete declarations, and then strip these declarations out, we can measure the capabilities of the type inference system to put them back. If no unique set of declarations can be inferred, we have a problem with the type system itself, whereas if a unique set of declarations exists, but the inferencing algorithm cannot always find it, then we have a problem with the inferencer.

The major problem in type inferencing has been how to deal with *polymorphism*, where the typing of a function symbol or value symbol is different in different contexts. Polymorphism is of great practical concern due to the success of object-oriented systems, which routinely manipulate higher-order objects such as function-producing functions, and these functions may have to deal with many different types.

The most successful type inferencing algorithm in use today is Milner's unification algorithm which can handle a certain amount of polymorphism. Probably the best exposition of this algorithm for our purposes is given in [Aho86,p.364], because it gives a unification algorithm for the type inferencing method in the section just following [Aho86,p.376]. Rather than give a detailed explanation of this algorithm which is better covered in that reference, we will make a few observations about how and why it works.

Unification is a process whereby a number of objects are first assumed to be distinct, and it is later discovered that some of them are actually identical. In the case of type inference, distinct mathematical variables are proposed for each program variable and temporary within a program, where the mathematical variables will each denote an unknown "type". Using the knowledge of the primitives of the programming language, we can prove that many of these variables are not distinct after all, but identical. Complications arise when higher-order types (functionals, data structures) are involved, because determining the "identity" of two types now involves a matching process.

Crucial to the success of Milner's polymorphic type inferencing process is the fact that when a polymorphic function type is to be represented, a <u>fresh new copy</u> of that function type is constructed (the exact semantics of "fresh new copy" are important to us and will be discussed later). Since the new copy is distinct from other instances of the function type, the type inferred in this instance can be different from that inferred in other instances.

We claim that this copying process is functionally identical to procedure integration ("inlining", "beta-substitution"), so far as the type inferencing algorithm is concerned, although it is not so computationally intensive as actual inlining. Thus, the Milner-style inferencing is still not "really" polymorphic, because the different instances of the function are no longer the "same" function.

We should not have to duplicate these remarks for data structures in functional languages, due to the ability of functional closures to simulate data structures. However, we will, because we are vitally interested in the detailed mechanics of type inferencing for data structures.

Data structures such as a Lisp-like *pair* are treated in a fashion similar to functions by Milner's type inference algorithm. Whenever a pair is constructed or used, it posits the possibility of a *new* pair—never before seen—and then utilizes other constraints to force identity when identity cannot be denied.

We now begin to see the possibility of using Milner-style type inferencing to determine sharing and containment relationships. Each new pair type node constructed during the execution of Milner-style type inferencing starts out by representing a separate (disjoint) set of run-time nodes. It is only when other constraints force two of these nodes to be the same (i.e., the sets potentially overlap), that the nodes are identified. Therefore, at the end of Milner-style type inferencing, the nodes which are not unified together represent disjoint sets of run-time nodes, assuming that we have seen the entire program. This last assumption is crucial, since unification starts out assuming that everything is different, and moves monotonically toward sameness. The only thing standing between a given program and universal oneness is the lack of any more constraints to apply.

Milner-style type inferencing therefore gives us the following very important information: each distinct datatype node represents a set of runtime nodes, which cannot be distinguished by the current level of analysis. Datatype nodes which are distinct represent disjoint sets of runtime nodes, while datatype nodes which are unified represent overlapping sets of runtime nodes. Milner-style type inferencing therefore defines a particular level of sharing and containment information which can be gotten relatively cheaply, but is still quite helpful in solving allocation and updating problems. For example, we show later that Milner-style type inferencing can infer that the top level of the result of Lisp's `append` function does *not* share with the top level of the first argument to `append`.

3. Insight into our Milner-style type-cum-storage-use analysis

Researchers in storage use inference have been forced to make *approximations* in the detail of their analysis in order to ensure that the analysis terminates within a reasonable period of time. One of the most common approximations in their analysis has been the classification of all runtime objects allocated by a particular "allocation site" within a program as being *equivalent*, because they were allocated by the same piece of program text—the *birthplace* [Ruggieri87, Chase87, Hederman88, Larus89]. However, this notion is not completely well-defined, because if one shifts one's attention to the allocation subroutine itself, then *all* nodes would be equivalent, and the analysis would become worthless. Our interpretation of this notion is that the allocation subroutine is always procedurally integrated (inlined), so that allocation points and program texts are in a one-to-one correspondence.

Consider, for example, a Lisp-like *pair* datatype. In making a distinction among the *pairs* that occur at run-time within a program based on where in the program text the pair was allocated, we are creating a new set of "datatypes" which is a refinement of the standard set of datatypes. We thus have pair_1, pair_2, pair_3, etc., as atomic datatypes, where the only difference between them is their birthplace. While this refined type system is larger than the original type system due to the additional distinctions, it is still tractable if the original type system was. This is because the program text is finite, and therefore the refined type system cannot be that much bigger than the original type system.

A similar approximation is made with respect to functions. While different levels of recursion can represent substantially different situations to be modeled, we are forced to curtail this type of excursion, else our analysis will become paralysis. Therefore, we utilize the same "textual" approach to approximation for functions that we used for data structures: if two functions share the same text but are different closures (i.e., the function text was closed with the values of its free variables at different execution times), then they are still the same "function", at least so far as our static modeling goes. Note that this approximation is not quite as restrictive as it appears, because we can always perform procedure integration (inlining) before starting our static analysis, after which each instance of the original procedure will be handled completely independently.

From an implementation point of view, this tagging of objects with their textual birthplace is not such a bad idea. After all, a compiler must choose a particular machine code expansion for each birthplace, so there is some possibility that different birthplaces can be handled differently. If subsequently this information can be utilized to optimize deallocation or in-place updating, then this birthplace distinction among datatypes was worthwhile. Conversely, it would be a lot of work for a compiler to make any *finer* distinction than birthplaces, because it would involve additional parametrization and branching in the allocation code. Finally, birthplace tagging seems to be a good approximation to what expert programmers do when they implement more intelligent allocation/deallocation than automatic garbage collection.

The subdivision of standard datatypes created by our birthplace tagging can be considered to produce *distinct* types, even though the same operations can be performed on them. Therefore, we have a new form of *polymorphism* which, if expressed as truly distinct types, cannot be handled by traditional Milner-style type inferencing algorithms. This is because while the program without birthplace tagging may be uniquely typed, the program with birthplace tagging may require the ability to handle *sets* of types. For example, the expression (**if** … **then** pair_1 **else** pair_2) has the *single* type "pair" before tagging, but the *set* of types {"pair_1","pair_2"} after tagging. Suzuki has elegantly extended Milner-style type inferencing to deal with such a situation in Smalltalk [Suzuki81], while the Russell compiler [Boehm86] utilizes another style of type inferencing with union types.

(We have attempted to model birthplace tagging using Ada *derived types* [Ada83], without success. While derived types in Ada are essentially copies of a type and all of its operations, Ada forces all derivations of a type to use the same storage collection, which removes the very distinction we were trying to make!)

4. An example

We will now show an example of how Milner-style type inference unwittingly produces sharing information. Consider the pure Lisp function `append`:

```
(defun append (x y)
  (if (null x) y
      (cons (car x) (append (cdr x) y))))
```

We would like to perform traditional Milner-style type inference on `append`. However, since pure Lisp has no standard type system, we give it a quasi-ML type system (as in [Wand84]), in which *lists*, for example, have arbitrary length, but all list elements must be of the same type. We must first assign types to the primitive constants in the program—namely `if`, `null`, `cons`, `car`, `cdr` (we indicate the type of an expression by the notation "<expression> : <type>", as in ML).

```
if:     boolean x 'E x 'E → 'E      ; 'E is a type variable.
null:   list('A) → boolean          ; 'A is a type variable.
cons:   'B x list('B) → list('B)    ; 'B is a type variable.
car:    list('C) → 'C               ; 'C is a type variable.
cdr:    list('D) → list('D)         ; 'D is a type variable.
```

A possible sequence of type inferences is as follows:

```
x: list('A)                             ; from null(x)
'A = 'C = 'D                            ; from (car x), (cdr x)
(car x): 'A
(cdr x): list('A)
(append (cdr x) y): list('A)            ; from defn of cons
y: list('A)
append: list('A) x list('A) → list('A)
```

Thus, `append` takes two argument lists and returns a result list and all three lists must have the same element type. This example shows traditional Milner-style type assignment at its best. We now look more carefully at the data structures created during the process of unification. Here we assume a "fast" unification algorithm similar to that found in [Aho86,p.376]. The important thing is that all variables 'Z share the same node, and that nodes like "list('Z)" are not *necessarily* the same, and start out being (usually) different.

We now redefine the types of the primitive functions, being more careful about different *instances* of nodes like "list('Z)". We distinguish these differences by means of subscripts. It is vitally important, however, that the instances are subscripted in accordance with the sharing semantics of the functions being typed, as we shall later see.

```
if:     boolean x 'E x 'E → 'E      ; no change
null:   list_1('A) → boolean
cons:   'B x list_2('B) → list_2('B)
car:    list_3('C) → 'C
cdr:    list_4('D) → list_4('D)
```

Now, when we do Milner-style unification, we can see the different instances getting unified.

```
y: 'E    ; from (if ...)
x: list_1('A)                           ; from (null x)
list_3('C) = list_1('A)                 ; from (car x)
'C = 'A
list_4('D) = list_1('A)                 ; from (cdr x)
'D = 'A
'B = 'C                                 ; from (cons ...)
(append ...): list_2('B)                ; from (cons ...)
'E = list_2('B)                         ; from (if ...) and result

To summarize:
'A = 'B = 'C = 'D
x: list_1('A) = list_3('C) = list_4('D)
y: 'E = list_2('B)
append: list_1('A) x list_2('A) → list_2('A)
```

As before, `append` takes as arguments two lists of the same type of elements, and returns a list of the same type of elements as its arguments. In addition, we know that the result of `append` must share with its second argument (even if it only shares the empty list `nil`), and may also share with its first argument if the first argument were sharing with the second argument already, as in

```
(let ((x '(a b c)))
  (append x x))
```

Thus we have shown is that while the *types* of the arguments and the results of the `append` function are all isomorphic, the result of `append` cannot contain cells from the top level of its first argument, unless the first and the second arguments were already sharing, based on constraints from *outside* of the `append` function.

Why did we type `cdr` as $\texttt{list}_4\texttt{('D)}\rightarrow\texttt{list}_4\texttt{('D)}$ instead of as $\texttt{list}_4\texttt{('D)}\rightarrow\texttt{list}_5\texttt{('D)}$? The reason is that `cdr` doesn't just give a *copy* of the rest of the list, it gives *the* rest of the list. In other words, the type of `cdr(cons(x,y))` cannot be an isomorphic copy of the type of `y`, but must be the type of `y` itself. In the relatively simple type system given here, this can only be arranged by making the type of the result of `cdr` *identical* to the type of its argument. (Note that the type of `car(cons(x,y))` is already identical to the type of `x`). Unfortunately, this means that we cannot distinguish different cells in the simple list `cons(1,cons(2,cons(3,nil)))`, because they all have *identical* types.

A curious feature of our ML-style type system is that `nil` is the polymorphic generator of new lists. In other words, since `cons` produces the type of its second argument, the only way to introduce a new list type is by using `nil`.

(Another type system might type `cons` as `'Ax'B→pair('A,'B)`, `car` as `pair('C,'D)→'C`, and `cdr` as `pair('E,'F)→'F`. Such a type system would also satisfy the requirements that the type of `car(cons(x,y))` is identically the same as the type of `x` and the type of `cdr(cons(x,y))` is identical to the type of `y`, and thereby distinguish the different cells in `cons(1,cons(2,cons(3,nil)))`, but this type system could not handle lists of indefinite length, because lists of different lengths could no longer be unified. Thus, the limitations of the type system itself reduce the potential resolution for our storage use inferencing.)

We can make the unification algorithm for sharing even more obvious (but less efficient and no more powerful), by including an additional *location* variable in addition to the *type* variables. In other words, we define the primitive functions as follows:

```
if: boolean x 'E x 'E
null: list('A,'L1) → boolean
cons: 'B x list('B,'L2) → list('B,'L2)
car: list('C,'L3) → 'C
cdr: list('D,'L4) → list('D,'L4)
```

Having done this, we perform traditional Milner-style type inference using unification to produce the following typing:

```
'A = 'B = 'C = 'D
'L1 = 'L3 = 'L4
'E = list('A,'L1)
append: list('A,'L1) x list('A,'L2) → list('A,'L2)
```

We must now provide an interpretation for these *location variables*. A location variable ranges over all locations. Two location variables *may* share if they are unified together. Our typing of the `append` program above provides the *most general unifier* this program, hence the most general typing. There may still be sharing between the first argument to `append` and the result, but this sharing would not be caused by `append` itself, but would be the result of some other constraint outside of `append`.

Where are the location *constants*? Presumably, location constants would be individual locations, or at least *classes* of locations, all of whom were allocated by the same textual program statement. However, such an approximation cannot be handled by the simple non-state-based type inference system of ML, because different conses would yield different location constants, which could not be unified. This means, for example, that it would be impossible to unify the two arms of the conditional `(if x (cons 3 4) (cons 3 4))`. However, by leaving the locations in the form of location *variables*, we lose resolution, because we can no longer track the different allocations separately, but we can still do unification.

One consequence of this unification approach to sharing is that in order to get the best information, we must use a more complex typing scheme, which allows for *unions* of types, and thus can handle the two arms of the above conditional correctly. [Suzuki81] describes such an approach, which should be able to handle the situation where the location constants for each allocation are a representation of location within the program text where the allocation is made.

5. Soundness

We do not yet have a proof of soundness for our scheme, but we do have some observations. Due to the nature of static typing, two objects cannot share unless they have identical or isomorphic types, a fact which has been used by compilers for years [Ada83]. Since our scheme produces only information of the form "x *may* share with y" and not "x *must* share with y", we need only show that if "x may share with y" at run-time, then x and y will be assigned the same identical type. Since the accurate determination of "x may share with y at run-time" is undecidable, we will compute a conservative approximation to this relationship. Therefore, we need only prove that if x and y have different (isomorphic but not identical) types, then x *cannot* share with y at run-time.

A lemma is needed which shows that for our scheme to go wrong, the type inference system must assume that a copy will be made, when in fact a value will actually be passed uncopied, and the result becomes an undetected alias. Such situations arise in several cases. Cells which are returned to the free list and reallocated may indeed "share", but this situation does not cause problems because it is no longer the "same" cell—only its storage was reused. Similarly, in-place aggregate updating can also result in undetected sharing, but this situation is identical to the previous one, except that we never bothered to return the object to the free list and reallocate it.

However, there are more interesting cases to consider. Lisp's `intern` function, for example, either returns its argument or another symbol with the same "print-name"; however, it does not create any new symbols. Proper typing requires that *all* symbols become aliased, since it is impossible to precisely model the sharing semantics of `intern` using simple types. The use of "memoization" to "cache" function values—a legitimate optimization in functional languages—offers a similar opportunity to avoid making copies, and thereby cause an undetected alias. Goto suggests the use of "hash consing" [Goto74] to implement data structures in functional languages so that the recursive `equal` function no longer has to recurse. Hash consing always produces the unique value with a particular set of selectors (e.g., car and cdr), so it always produces a unique representation for every structured value, no matter how complex. In such a system "copying" has no meaning, since a copy operation will always return the same (pointer) value, and therefore there is a possibility that our scheme can "go wrong" in this case. The intuitive "fix" in all of these cases is to match the run-time semantics with an appropriate version of "fresh new copy" in the Milner-style type inference algorithm. In such cases, "stale" might be better than "fresh".

Complications such as these have so far kept us from a proof of soundness.

6. Applications

Our scheme can reclaim all of the garbage from *quicksort* [Inoue88], if one correctly types the polymorphic recursive function `qs`, below. Traditional ML type inference cannot do this, but ML+ [Kfoury88] should be able to.

```
(defun qs(x)
  ; qs: list1(integer)→list2(integer)
  (if (null x) nil
    (append (qs1 (low (cdr x) (car x)))     ; append defined above.
            (cons (car x)
                  (qs2 (high (cdr x) (car x)))))))

(defun high(x i)
  ; high: list3(integer) x integer→list4(integer)
  (cond ((null x) nil)
        ((< (car x) i) (high (cdr x) i))
        (t (cons (car x) (high (cdr x) i)))))

(defun low(x i)
  ; low: list5(integer) x integer→list6(integer)
  (cond ((null x) nil)
        ((>= (car x) i) (low (cdr x) i))
        (t (cons (car x) (low (cdr x) i)))))
```

This can be done because our typing shows that `qs`, `high` and `low` all copy their arguments, and hence their results do not share with their arguments. Since the intermediate results from calling `low`, `high` and $\texttt{qs}_1$ do not share with the result of `qs`, they can all be garbage collected by `qs` before it returns.

Our scheme can do destructive aggregate updating [Bloss89].

```
upd: seq1('E) x integer x 'E → seq2('E)
; upd(a,i,x) = a', s.t. a'[i]=x and a'[j]=a[j], j≠i.

(defun ntimesx(n x)
  ; Create a new array of length n initialized to x.
  ; ntimesx: integer x 'E → seq('E)
  (labels
    ((init (a i x)
       ; init: seq3('E) x integer x 'E → seq4('E)
       (if (zerop i) a
         (init (upd a i x) (1- i) x))))
    (init (make-array n) (1- n) x)))
```

We can destructive updates to a, and hence avoid $O(n^2)$ behavior, because a doesn't share with anything else.

We do not claim that the storage structure created by Milner-style type inference is ideal for answering all such questions, but that it does have the appropriate information content. This insight should lead to modified versions of Milner-style type inference in which better storage structures are built.

7. Comparison with other techniques

Our method produces information about the possibility of a pointer in one "type" (set of objects) pointing to an object in another "type" at run-time. It does this by representing these sets as distinct objects which can point to one another. In this way, our method is analogous to the graph structures used by some researchers to represent sharing and containment information [Ruggieri87, Chase87]. Insofar as these researchers have deviated from our assumption of putting textual birthplaces in a 1-1 correspondence with sets of run-time objects, they can obtain more powerful results, because such deviations allow finer distinctions to be made among the various run-time objects than does our representation [Chase87, Larus89]. However, it is very difficult to characterize these finer distinctions, because they attempt to separately track, for example, the objects created at different levels of a recursion which can be unbounded, and therefore most researchers require that the analysis be cut off at some finite arbitrary limit (e.g., "k-limited" [Jones79]) in order to guarantee that the analysis will terminate. We believe that while separately tracking such objects is required to prove the correctness of some pointer-manipulating programs [Pleban81, Mason86], this tracking is too difficult and too expensive to include as a standard optimizing technique within a compiler. Furthermore, a programmer can often obtain the same results himself by explicitly instructing the compiler to "inline" the recursive function k times before our analysis is performed.

Chase [Chase87] also extends the graphical technique to handle certain kinds of side-effects, and is therefore more powerful than our technique, which is restricted to functional languages. However, the introduction of side-effects greatly clutters his model, and increases the computational demands of his analysis.

Researchers have also used explicit sets and set operations to compute sharing and containment information [Schwartz75, Inoue88], but we believe that our analysis subsumes theirs.

For the aggregate update problem, graphical analyses and static reference count analyses [Barth77, Hudak86, Hederman88] are *complementary*, in the sense that static reference counting can sometimes produce sharper information than any level of graphical analysis (e.g., on "sorting" and "swapping" programs [Hedermann88]), and vice versa. Therefore, one could perform a static reference count analysis in addition to our analysis and gain sharper information.

8. Complexity

The complexity of our analysis should be essentially the same as that for Milner-style type inferencing, which has been shown to be exponential [Kanellakis89][Mairson90]. However, since this type inferencer has been in use for a decade with few complaints about its speed, we assume that its worst case behavior is rare, although Mairson [Mairson90] shows a 5-line ML example which after 2 minutes of processor time, 60 megabytes of memory, and 173 printed pages of output, was still not typed! If we improve the resolution of the inference by using Suzuki-style type inference [Suzuki81], then the complexity will grow, since Suzuki requires an additional transitive closure operation whose size is on the order of the size of the program text, and the Warshall transitive closure algorithm [Baase78] (the most efficient "real" algorithm) is of cubic complexity.

9. Conclusions

Unification has been used by Milner and others for automatically inferring datatypes in a compiler for a programming language. Fast algorithms for unification utilize clever "structure-sharing" algorithms to obtain their efficiency.

Graphs of various sorts have been used by many researchers in order to statically determine at compile time some of the *sharing* properties of structures created at run-time [Chase87, Ruggieri87, Hederman88, Larus89]. These graphs are used for purposes such as detecting opportunities for in-place updating in functional languages, detecting opportunities for converting heap allocation into stack or static allocation, detecting aliases which interfere with traditional optimizations, and detecting dependencies during parallelization.

We have shown that the clever structure-sharing unification algorithms for Milner-style type inference *already produce sharing information*, which can then be used for the indicated storage use optimizations. Alternatively, we can make storage aliasing explicit during the unification algorithm, in order to extract even more sharing information.

Using this insight, we have developed an elegant algorithm which performs type inferencing and sharing inferencing at the same time. This algorithm is powerful enough to handle most of the storage use problems for which graphical techniques have been advocated.

We discovered the unification method of sharing inferencing while we were studying the implementation of Milner-style type inferencing described in [Aho86,p.364], and simultaneously studying the "storage containment graphs" of David Chase [Chase87]. Both of these algorithms compute similar structures! An additional insight came from the observations of several researchers [Jones79, Pleban81, Larus89, Inoue88], that the alias problem can be solved by describing the possible pointer paths as *regular expressions*, which have nice closure properties under intersection, union and complement. Since storage containment graphs (and other similar finite graphs defined by other researchers) are essentially non-deterministic finite state machines for recognizing these regular expressions, and since unification can be seen [Aho86,p.388] as a finite-state automaton equivalence tester, it seemed possible to solve all of these problems with the same machinery.

The information produced by our analysis is essentially equivalent to the information produced by the analysis of [Inoue88] for "pure" (functional) Lisp, except that our algorithm is a good deal simpler and easier to understand than theirs. Since the information is equivalent, the storage use optimizations they perform should also be implementable using our algorithm.

Our analysis does not easily extend to non-functional languages, because it is much more difficult to capture the realm of sharing and containment relations that could occur during run-time by using static analysis on programs which can alter such relationships through assignment. Chase addresses this more general problem [Chase87]. However, the complexity of his analysis for non-functional languages may make it impractical, while our analysis is certainly practical for functional languages, because most implementations are already unwittingly doing it—as we have shown!

10. Acknowledgements

We are indebted to the reviewers for their help in improving this paper—especially to those who pointed out embarrassing errors in the first draft. We are also indebted to the U.S. Department of Energy for their support of this research (Contract #DE-AC03-88ER80663), and to David Chase for his patient explanation of his thesis work to me. Finally, we are indebted to Andrew Appel and Peter Lee, who told us about the elegance of ML type inference.

11. References

Ada. *Reference Manual for the Ada® Programming Language*. American National Standards Institute, New York, 1983.

Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools.* Addision-Wesley, Reading, MA, 1986.

Appel, A.W., Ellis, J.R., and Li, K. "Real-time concurrent garbage collection on stock multiprocessors". *SIGPLAN PLDI*, June, 1988.

Baase, S. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, Reading, MA, 1978.

Baker, H.G. "List processing in real time on a serial computer". *CACM 21*,4 (April 1978),280-294.

Baker, H.G. "The Nimble Type Inferencer for Common Lisp-84". Nimble Computer Corporation, 1990, *in preparation*.

Barth, J.M. "Shifting garbage collection overhead to compile time". *CACM 20*,7 (July 1977),513-518.

Bloss, A. "Update Analysis and the Efficient Implementation of Functional Aggregates". *4'th ACM/IFIP Conf. on Funct. Prog. and Comp. Arch.*, London, Sept. 1989, 26-38.

Boehm, H.-J., and Demers, A. "Implementing Russell". *ACM Sigplan '86 Symp. on Compiler Constr., Sigplan Notices 21*,7 (July 1986),186-195.

Chase, David R. *Garbage Collection and Other Optimizations*. Ph.D. Thesis, Rice University, August, 1987.

Goto, Eiichi. "Monocopy and Associative Algorithms in an Extended Lisp". Info. Sci. Lab., Univ. of Tokyo, 1974.

Hederman, Lucy. *Compile Time Garbage Collection*. M.S. Thesis, Rice Univ. Comp. Sci. Dept., Sept. 1988.

Horowitz, S., Pfeiffer, P., and Reps, T. "Dependence Analysis for Pointer Variables". *ACM Sigplan '89 Conf. on Prog. Lang. Design and Impl.*, Sigplan Notices 24,7 (July 1989),28-40.

Hudak, P., and Bloss, A. "The aggregate update problem in functional programming systems". *12'th ACM POPL*, January, 1985.

Hudak, P. "A Semantic Model of Reference Counting and its Abstraction". *1986 ACM Lisp and Functional Programming Conference*, Cambridge, MA,351-363.

Ichbiah, J.D., et al. "Rationale for the Design of the Ada Programming Language". A*CM Sigplan Notices 14,*6 (June 1979), Part B.

Inoue, K, Seki, H., and Yagi, H. "Analysis of functional programs to detect run-time garbage cells". *ACM TOPLAS 10*,4 (Oct. 1988), 555-578.

Jones, N.D., and Muchnick, S.S. "Flow Analysis and Optimization of Lisp-like Structures". *6'th ACM POPL*, Jan. 1979,244-256.

Jones, N.D., and Muchnick, S.S. "A flexible approach to interprocedural data flow analysis and programs with recursive data structures". *9'th ACM POPL*, 1982,66-74.

Jones, S.B., and Le Metayer, D. "Compile-time garbage collection by sharing analysis". *ACM Func. Prog. Langs. and Comp. Arch. (FPCA)*, 1989, 54-74.

Kanellakis, P.C., and Mitchell, J.C. "Polymorphic unification and ML typing". *16'th ACM POPL*, Jan. 1989,105-115.

Kfoury, A.J., Tiuryn, J., and Urzyczyn, P. "A Proper Extension of ML with an Effective Type-Assignment". *Proc. 15'th ACM POPL*, Jan. 1988,58-69.

Larus, James Richard. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. Ph.D. Thesis, UC Berkeley, also published as Rep. No. UCB/CSD/89/502, May, 1989.

Lieberman, H., and Hewitt, C. "A real-time garbage collector based on the lifetime of objects". *CACM 26*,6 (June 1983),419-429.

Mairson, H.G. "Deciding ML Typability is Complete for Deterministic Exponential Time". *17'th ACM POPL*, January 1990, 382-401.

Mason, Ian A. *The Semantics of Destructive Lisp*. Center for the Study of Language and Information, Stanford, 1986.

Milner, R. "A theory of type polymorphism in programming". *JCSS 17*,3 (1978),348-375.

Mitchell, J.C. "Coercion and Type Inference". *11'th ACM POPL*, 1984, 175-185.

Moon, D. "Garbage collection in a large Lisp system". *ACM Lisp and Functional Programming Conf.*, 1984,235-246.

Muchnik, S. S., and Jones, N.D. "Binding time optimizations in programming languages: some thoughts toward the design of an ideal language". *ACM POPL*, 1976.

Muchnik, S., and Jones, N. "Flow analysis and optimization of LISP-like structures". In *Program Flow Analysis: Theory and Applications*, by the same authors, Prentice-Hall, Englewood Cliffs, NJ, 1981.

Peyton-Jones, S.L. *The Implementation of Functional Programming Languages*. Prentice-Hall, NY, 1987.

Pleban, Uwe F. *Preexecution Analysis Based on Denotational Semantics*. Ph.D. Thesis, University of Kansas, 1981.

Ruggieri, Christina. *Dynamic Memory Allocation Techniques Based on the Lifetimes of Objects*. Ph.D. Thesis, Purdue University, August, 1987.

Ruggieri, C. and Murtagh, T. "Lifetime analysis of dynamically allocated objects". *15'th ACM POPL*, January, 1988.

Schwartz, J.T. "Optimization of very high level languages, Part II: Deducing relationships of inclusion and membership". *Computer Languages 1*,3 (1975),197-218.

Steele, G.L., Jr. *Rabbit: A compiler for Scheme*. AI Memo 474, MIT, May 1978.

Suzuki, N. "Inferring types in Smalltalk". *8'th ACM POPL*, 1981,p.187-199.

Ungar, D. "Generation scavenging: A non-disruptive high performance storage reclamation algorithm". *ACM Software Eng. Symp. on Prac. Software Dev. Envs., SIGPLAN Notices 19*,5 (May 1984),157-167.

Wand, M. "A Semantic Prototyping System". Proc. of ACM Sigplan '84 Symp. on Compiler Constr., Sigplan Notices 19,6 (June 1984),213-221.

Wand, M., and O'Keefe, P. "On the Complexity of Type Inference with Coercion". *ACM Func. Prog. Langs. and Comp. Arch. (FPCA)*, 1989, 293-297.