LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1993

A Practical Approach to Type Inference for EuLisp

ANDREAS KIND* (*andreas.kind@isst.fhg.de*)

HORST FRIEDRICH (*horst.friedrich@isst.fhg.de*)

Fraunhofer Institute for Software Engineering and Systems Engineering (ISST), Kurstrasse 33, 10117 Berlin, Germany

Keywords: Type Inference, Lisp, Compilation, Unification.

Abstract. Lisp applications need to show a reasonable cost-benefit relationship between the offered expressiveness and their demand for storage and run-time. Drawbacks in efficiency, apparent in Lisp as a dynamically typed programming language, can be avoided by optimizations. Statically inferred type information can be decisive for the success of these optimizations.

This paper describes a practical approach to type inference realized in a module and application compiler for EuLisp. The approach is partly related to Milner-style polymorphic type inference, but differs by describing functions with *generic type schemes*. Dependencies between argument and result types can be expressed more precisely by using generic type schemes of several lines than by using the common one-line type schemes. Generic type schemes contain types of a refined complementary lattice and bounded type variables. Besides standard and defined types so-called *strategic types* (e.g. singleton, zero, number-list) are combined into the type lattice. Local, global and control flow inference using generic type schemes with refined types generate precise typings of defined functions. Due to module compilation, inferred type schemes of exported functions can be stored in export interfaces, so they may be reused when imported elsewhere.

1. Introduction

Static type checking makes use of the declarative information available in a program. On the one hand, programming languages with required type declarations have advantages over declaration-free languages:

- during compile-time all type inconsistencies can be uncovered,
- compilers are able to create more efficient code by doing optimizations,

*This work was supported by the German Federal Ministry for Research and Technology (BMFT) within the joint project APPLY. The partners in this project are the Christian Albrechts University Kiel, the Fraunhofer Institute for Software Engineering and Systems Engineering (ISST), the German National Research Centre for Computer Science (GMD), and VW-GEDAS.

- static type information can be used to improve storage management,
- enhanced program documentation is achieved.

On the other hand, required declarations lead to a loss in flexibility and expressiveness, because program structures are difficult to reuse and extend—depending on the limitations and flexibility of the type scheme.

Lisp as a dynamically typed programming language offers flexibility and expressiveness. This quality comes with potentially expensive run-time type checking, much of which is unnecessary. The polymorphism of a function may be limited in the context of another function. For example, + is generally defined on all number types. Consider a call to + where both arguments are the results of a function that computes the length of lists, then the result of + can be inferred to be a positive integer.

Steenkiste and Hennessy [13] point out that type computations for Lisp applications increase the execution time by 25%, on average. For individual applications this value may lie between 6% and 88%. The checking of list operators, in particular, can constitute a major part of the execution time. These drawbacks in efficiency, apparent in Lisp as a dynamically typed programming language, can be reduced by using suitable approaches to type inference without sacrificing Lisp's flexibility and expressiveness.

The results presented here were obtained as part of the APPLY project. The aim of the project is to develop a modern and practical Lisp system [3]. Efficiency and integration can only be reached in line with corresponding demands on language. These include the following features, that are complied with by the Lisp dialect EuLisp [11].

- separately compiled modules,
- clear separation between compile-time and run-time,
- far-reaching static analysis,
- separation of language from development environment.

Hence, we decided to build a module and application compiler for EuLisp. In order to achieve compiled modules and applications with efficient run-time behaviour a practicable type inference system is integrated in the compiler. The advantages of static type inference are:

- reduction of dynamic type checks,
- increased use of machine data types instead of program data types,

- greater chances for further optimizations (e.g. inlining, dead code elimination).

The combination of EuLisp and generic type schemes with refined types allows us to improve on previous work on type inference for Common-Lisp. *TICL* a type inference system developed by Ma and Kessler [8] generally achieves 20% speed improvement. However, reanalysis slows the inference process when recursive functions must be handled, or when a defined function is analysed before those functions that use it are analysed.

A type inference approach proposed by Baker [1] inspired us to use refined types. But the need to use the costly Kaplan/Ullman fixed-point algorithm makes this approach less attractive. Neither *TICL* nor Baker's approach handle the notion of typed lists.

A global tagging optimization for Scheme, proposed by Henglein [5], eliminates 60–95% of tag handling operations in non-numerical code by compile-time inference. This approach to type inference does not concern itself with refined types or module compilation.

2. General Approaches to Type Inference

The general approaches to data type inference depend on the type discipline of the programming language in question. Lexical monomorphic languages (e.g., Pascal) enable the direct derivation of types for all expressions. Types have to be explicitly assigned to all constants, variables and functions. A simple recursive algorithm can then be used to determine the type of an expression from the types of its subexpressions.

Programming languages with polymorphic type disciplines allow the introduction of type variables into type expressions in order to achieve greater flexibility. The types of expressions can be derived statically in these languages from:

- available type declarations,
- type descriptions of the standard functions,
- contextual type information.

There are two main approaches to static type inference: Milner-style unification and Kaplan/Ullman fixed-point iteration. We discuss each of these in turn.

2.1. Milner-Style Unification

Due to the type discipline of ML [10, 4] a static typing of polymorphic functions is possible. This approach to type inference tries to relate the types of all language expressions to each other via type variables. The relations of program structures are reflected in equations of type expressions which, as suggested by Milner [9], can be resolved by *unification* [12]. Full static typing is achievable by the constraints on the ML type discipline: variables and structure components are each limited to single types (no side-effects of types) and explicit type declarations are sometimes required.

2.2. Kaplan/Ullman Fixed-Point Iteration

The second approach to static inference of data types is attributable to a proposal by Jones and Muchnick [6]. Kaplan and Ullman [7] refined this idea to obtain more exact type information for given program statements.

In the approach suggested by Kaplan and Ullman, programs are modelled as directed graphs, with nodes representing assignments and edges representing direct control flow relations between assignments. The possible links between program variables $(x_1, \cdots, x_k, y)$ are considered before and after the execution of an assignment (Q).

$$\begin{array}{c} \cdots \\ \downarrow \\ Q : \boxed{y \leftarrow op(x_1, \cdots, x_k)} \\ \downarrow \\ \cdots \end{array}$$

Forward and backward analyses are iterated over the program nodes to determine as precise a type as possible for each program variable. *Forward analysis* infers type information *after* execution of an assignment from type information on program variables available *before* the execution of assignment. For example, this may imply that the result type is inferred from argument types of a function application. *Backward analysis* makes inferences against the direction of control flow. Type information on program variables *before* execution of an assignment is inferred from information available *after* execution of an assignment due to applications of functions whose possible argument and result types are known.

For forward and backward analyses, type descriptions of the standard functions are used together with a type lattice adapted to the type system of the programming language. Type descriptions of standard functions are realized in the Kaplan/Ullman approach as so-called T-functions, where argument types are associated with the corresponding result types, and combinations of argument and result types are associated with the types of selected arguments. In order to determine sharp type information on

program expressions a fixed-point algorithm is applied. Alternating forward and backward analyses are applied until no further refinement is obtained. The result of the forward analysis is used as upper bound for the backward analysis, and vice versa on successive iterations.

3. Characteristics of the new Approach

As an integral part of the compiler, the type inference system receives as input the source modules processed into an annotated abstract syntax graph together with type information about the imports. The type inference system adds inferred type information to the abstract syntax graph and generates type descriptions for export. Figure 1 illustrates the organization of the type inference system.

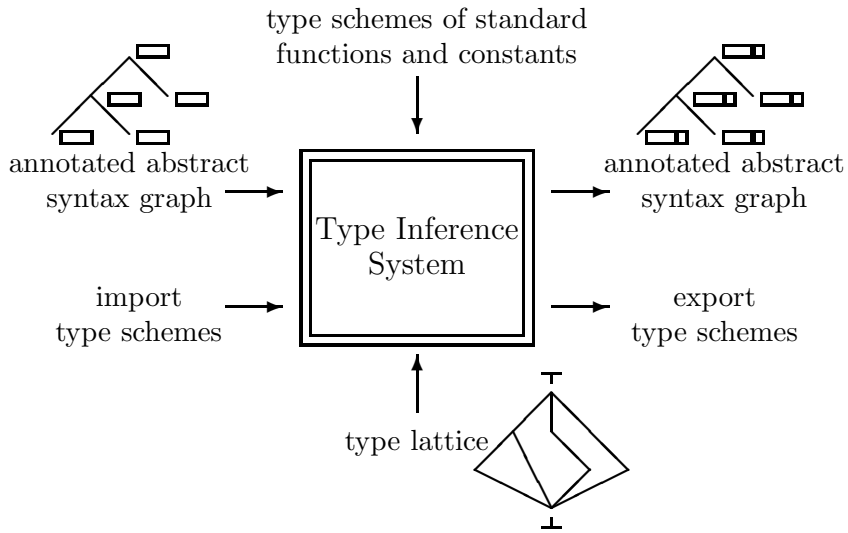


Figure 1: Type inference as integral part of the compiler

The approach follows that of Beer [2] in doing practical type inference by separating the realistic inferences from the unrealistic ones. This is the reason why for recursive functions we do not attempt to achieve as precise types as possible such as by use of Kaplan/Ullman fixed-point iteration.

Milner-style unification cannot be used for languages with side-effects on types [8]. Unlike ML, EuLisp provides polymorphic reference types (i.e. values of different types can be assigned to structure components) and has no need of lexical typing declarations. Nevertheless, the approach presented

here uses a modified unification algorithm to infer type schemes for defined functions without the need for expensive iterative analysis. In order not to give up the optimization of list operations, we distinguish monomorphic lists from polymorphic lists. Inference can be extended to the elements of monomorphic lists, but no other higher order data types are supported.

The following features characterize our approach:

refined type lattice: The success of type inference depends critically on getting sharp type information from standard functions and constants. That is why the refined lattice type contains more than just the standard and defined types. Refined types allow us to describe standard functions and constants more precisely. The type lattice is complementary, i.e. besides the lattice operations of union and intersection the complement of each lattice type is also defined.

generic type schemes: To describe the potential argument and result types of polymorphic functions, generic type schemes are used. The schemes contain lattice types and bounded type variables. Generic type schemes of standard functions are predefined; for defined functions they are inferred by a modified unification algorithm.

bounded type variables: Dependencies between argument types and result types are expressed in type schemes with type variables. The values that may be assigned to a type variable can be limited to a subset of all lattice types.

singleton types: The type lattice allows us to handle types with only one value (singleton types) specially. By means of singleton types, equality predicates, in particular, can be described more precisely.

control flow inference: Generic type schemes assist with control flow inferencing. Particular lines of the type scheme are assigned to each program branch at `if` or `cond`.

global inference: The language design of EuLisp assists in the problem of inferring global type information by providing encapsulation with modules[1]. Type information of functions, variables and constants is associated with definite parts of a program determined by the import and export interfaces. This reduces the computational cost of statically inferring global type information. For example, knowing all calls to a defined function enables us to infer the type scheme of the function. This global inference can be finalized with the analysis of

[1] Although we note that objects can also escape from modules by means of the class hierarchy and, in particular, methods on generic functions defined elsewhere.

the module, if the function is not named in the export interface. In this case, there can be no calls from outside the module, so that it is generally not necessary to keep track of all function calls over the entire application.

persistent type information: Compilation of EuLisp means the compilation of individual EuLisp modules by using import and export interfaces of already compiled modules. In order to reuse inferred type information of a compiled module, type descriptions of exported functions, variables and constants are added to the export interface.

3.1. The Type Lattice

This type inference approach makes use of a *complementary lattice*:

$$L := (T, \sqcup, \sqcap, \overline{\ })$$

with a non-empty set of types T and operations $\sqcup$ and $\sqcap$. The operations are commutative, associative and satisfy

$$\tau_1 \sqcup (\tau_1 \sqcap \tau_2) = \tau_1 \text{ and } \tau_1 \sqcap (\tau_1 \sqcup \tau_2) = \tau_1 \ for\ \tau_1, \tau_2 \in T.$$

The lattice is complementary, meaning that for every $\tau \in T$ there is at least one complement type $\overline{\tau} \in T$. The set of types can be divided into subsets:

$T :=$ Standard-Type $\cup$ Defined-Type $\cup$ Strategic-Type.
Standard-Type := {`<object>`, `<character>`, `<null>`, `<number>`, $\cdots$}.
Defined-Type := defined EuLisp structures and classes.
Strategic-Type:= {`singleton`, `zero`, `one`, `list`, `sy-list`, `fpi-list`, $\cdots$}.

The lattice types $\sqcup\tau_i$ and $\sqcap\tau_i$ for all lattice types τ_i are designated $\top$ and $\bot$, respectively. Instead of formally defining an order relation on all lattice types, we use Figures 2 and 3 to illustrate the lattice structure.

All primitive types handled inside a module can be determined at the start of the inference. Together with all types which can be constructed by the lattice operations they form a finite set L of lattice types. The type lattice thus has a finite number of elements. Bit codes are assigned to every element in order to implement the operations $\sqcup$, $\sqcap$ and $\overline{\ }$ as fast low-level bit operations (`and`, `ior`, `xor`). Expensive traversal of the lattice to compute union, intersection and complements of lattice types can thus be avoided.

3.2. Generic Type Schemes

Following Milner's theory of type polymorphism [9] type variables (α) are used to express constraints between argument types or between argument types and the result type of a function. In general, type variables stand for

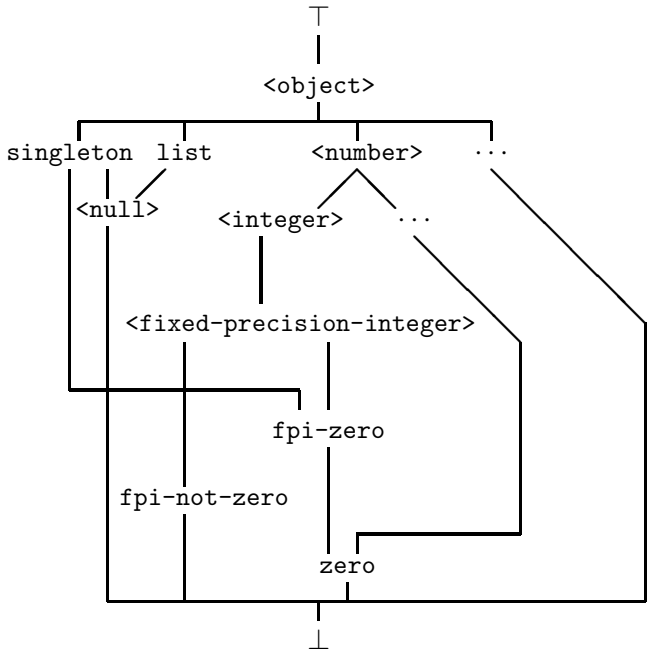


Figure 2: Standard and strategic types as part of the type lattice (a)

any lattice type, but they can be restricted to a subset of them, which is denoted by writing the upper bound type as a superscript. For example, the type variable $\alpha^{\texttt{<number>}}$ denotes all `<number>` types. This restriction of a type variable can be interpreted as if the variable was bound to the specified type.

$$\alpha^{\tau} \text{ may thus be written as } \alpha = \tau \text{ with } \tau \in L$$

Although the notion of equations is more common in connection with unification, superscripts are used to provide more readable type descriptions.

To track the full polymorphic capacity of functions and constants, the commonly used one-line type schemes are extended to *generic type schemes*:

$$\begin{array}{ccccccc} \delta_{arg_{11}} & \times & \cdots & \times & \delta_{arg_{1k}} & \rightarrow & \delta_{result_1} \\ \vdots & & \vdots & & \vdots & & \vdots \\ \delta_{arg_{n1}} & \times & \cdots & \times & \delta_{arg_{nk}} & \rightarrow & \delta_{result_n} \end{array}$$

A generic type scheme contains $n \geq 1$ lines each with fresh copies of type variables and $\delta := \tau \mid \alpha^{\tau}$, $\tau \in L$. By using these generic type schemes with

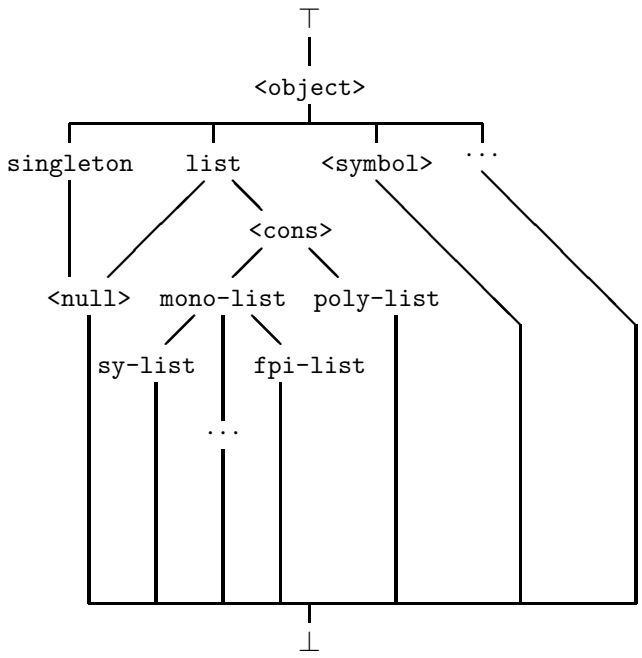


Figure 3: Standard and strategic types as part of the type lattice (b)

bounded type variables, dependencies between argument types and result types can be expressed more precisely. For functions, k designates the arity; constants are described by schemes with $k = 0$.

In the lattice shown in Figure 2, zero is extracted from all numerical subtypes and combined into the singleton type `zero`. The following generic type scheme is used for the standard predicate *zerop*:

zerop		
`zero`	→	$\overline{\texttt{<null>}}$
$\overline{\texttt{zero}} \sqcap$ `<number>`	→	`<null>`

The first line means that if the function *zerop* is applied to an argument of type `zero` the result will not be `()`, i.e. the complement type of `<null>`. The other descriptor means that if it is applied to an argument of type `<number>`, but which is not `zero`, the result will be `()` of type `<null>`. Separating the two cases enables us to perform control flow inferencing when *zerop* is used as the predicate function at a branch. Consider the

defined function *foo* using the predicate function *integerp*:

```
(defun foo (x y)
  (if (integerp x)
      (if (integerp y)
          x
        nil)
    nil))
```

Dependencies between argument and result types can be found where type variables occur inside the inferred generic type scheme.

foo				
$\alpha^{\texttt{<integer>}}$	$\times$	`<integer>`	$\rightarrow$	$\alpha^{\texttt{<integer>}}$
`<integer>`	$\times$	$\overline{\texttt{<integer>}}$	$\rightarrow$	`<null>`
$\overline{\texttt{<integer>}}$	$\times$	$\top$	$\rightarrow$	`<null>`

The type scheme for *integerp* is given in the appendix together with those for some other standard functions.

4. The Inference Routine

The inference routine begins analyzing function bodies by first doing local inferences, and second doing global inference by reducing locally inferred function schemes to type schemes that match for all known function calls of an application. A function has unknown calls if it is either exported or assigned to a variable (i.e. the function has to be translated into a closure).

During the analyses of function bodies, a type scheme must be available for every function application. If a called function does not have a type scheme, the analysis of the function being processed is suspended until a type scheme for the called function has been inferred.

When processing a function body, the incoming type information concerning argument types and the constraints defined by the type scheme of the function are unified for each function call in the body. The resulting type information is passed to later function calls. The incoming type information of a function call is called the *actual type constraints* of the called function. After unification of actual type constraints and each line of the formal type scheme, new actual type constraints are available, and these are used for unification with other function calls.

In general, the incoming type information consists of a set of actual type constraints, because for every line of the type scheme of the previously

called function new constraints can arise. The type scheme of each called function has to be unified with all the actual type constraints, but not all combinations of actual type constraints and lines of the type scheme can be unified successfully. That is to say, a function is not necessarily applicable to all incoming argument and result types. The number of new actual type constraints depends on the polymorphism of the called function and is at most the product of the number of actual type constraints and the number of lines in the type scheme.

Returning to the function *foo* given in section 3.2, we define a new function *bar* to illustrate the steps of unification.

```
(defun bar (x y)
  (let ((u (foo x y))
        (v (foo y x)))
    (if u v nil)))
```

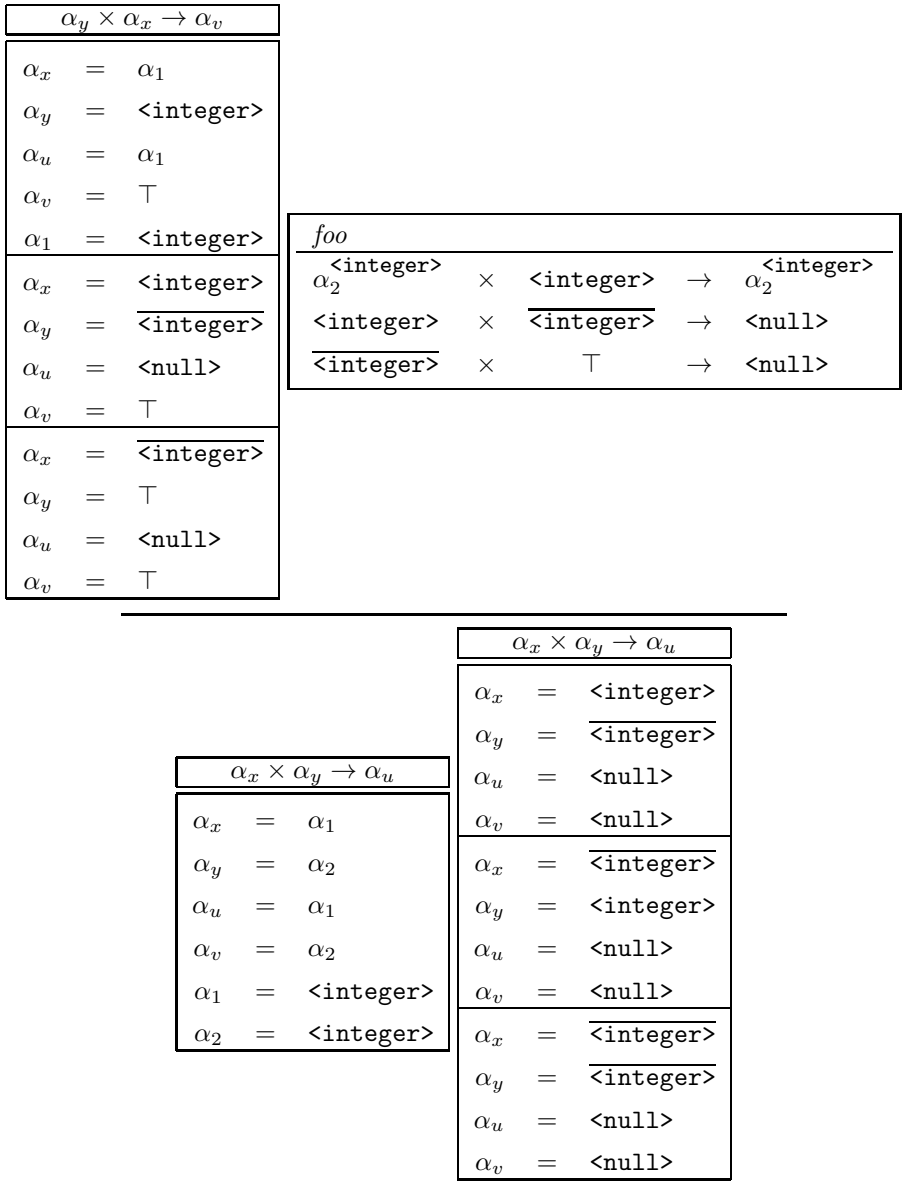
The process of local inference starts unifying the initial type constraints with the generic type scheme of *foo* to a set of new type constraints.

$\alpha_x \times \alpha_y \to \alpha_u$		
α_x	$=$	$\top$
α_y	$=$	$\top$
α_u	$=$	$\top$

foo				
α_1 <integer>	$\times$	<integer>	$\to$	α_1 <integer>
<integer>	$\times$	$\overline{\texttt{<integer>}}$	$\to$	<null>
$\overline{\texttt{<integer>}}$	$\times$	$\top$	$\to$	<null>

$\alpha_x \times \alpha_y \to \alpha_u$		
α_x	$=$	α_1
α_y	$=$	<integer>
α_u	$=$	α_1
α_1	$=$	<integer>
α_x	$=$	<integer>
α_y	$=$	$\overline{\texttt{<integer>}}$
α_u	$=$	<null>
α_x	$=$	$\overline{\texttt{<integer>}}$
α_y	$=$	$\top$
α_u	$=$	<null>

The function *foo* is applied once again but with arguments exchanged. The actual type constraints are those refined by the first call.



Afterwards, control flow inference selects the type constraints with $\alpha_u =$ <null> to pass into the then-case and those with $\alpha_u = \overline{\text{<null>}}$ to pass into the else-case. The result type can be determined in the then-case as <null> and in the else-case as the type of α_u. The type scheme of *bar* can now be given, after all unnecessary variables have been eliminated.

bar				
`<integer>`	$\times$	$\alpha^{\texttt{<integer>}}$	$\rightarrow$	$\alpha^{\texttt{<integer>}}$
`<integer>`	$\times$	$\overline{\texttt{<integer>}}$	$\rightarrow$	`<null>`
$\overline{\texttt{<integer>}}$	$\times$	`<integer>`	$\rightarrow$	`<null>`
$\overline{\texttt{<integer>}}$	$\times$	$\overline{\texttt{<integer>}}$	$\rightarrow$	`<null>`

The type scheme differentiates between different argument and result type ranges for *bar*. The first line reflects that when *bar* is called with two integer values, the result value is of the same type as the type of the second argument. If *bar* is called with a second argument of type `<fixed-precision-integer>` then the result is also of this type, provided that the first argument is an arbitrary integer value.

5. Monomorphic and Polymorphic Lists

Type information on global variables and structure components has to be treated carefully, because side effects on these components cannot be detected in general. However, redundant type checks in list operations are reduced for special kinds of lists. Thus, we distinguish monomorphic lists (`mono-list`) and polymorphic lists (`poly-list`)—see Figure 3. Monomorphic lists contain elements with the same type, for example `<symbol>`, `<number>` or `<cons>`. Figure 3 shows two kinds of monomorphic list types and their use in type schemes is shown in the appendix. When *cons* is used to add an item to a list, a monomorphic list may change to a polymorphic list if the new item is not of the same class as the rest of the list.

In some cases, type information inferred earlier on compound types may become invalid, because it is difficult to track all structure updates. To reduce the side effects that arise from update functions, all monomorphic lists are given a time stamp. If a function is called which is known to modify lists, all previously inferred polymorphic lists are subsequently treated as `<cons>` types.

Monomorphic numeric list types can be subdivided further into lists containing elements of type `<fixed-precision-integer>` $\sqcup$ `<single-float>` or `<fixed-precision-integer>` $\sqcup$ `<single-float>` $\sqcup$ `<double-float>`. Generic arithmetic operations can thus be optimized much better, because they often operate on lists of number lists using coercions to view the lists as monomorphic.

6. Recursive Functions

During local inference, a type scheme should be available for every function call, but if there is not, analysis continues with the rest of the body. This strategy does not work for recursive calls. When a type scheme is needed for a function that is being analysed, the default type scheme:

$$\top \times \cdots \times \top \to \top$$

is used. After finishing all pending local inferences, the functions involved in the recursion are analysed *once* again to specialize the type schemes. In order to achieve a practical type inference system we are able to do the analysis of recursion without a fixed-point iteration. An analysis with, for example, the Kaplan/Ullman algorithm, would be very expensive and would not be acceptable in a practical compiler for large applications.

The inferred generic type scheme of *length*, a function to compute the length of arbitrary lists, shows that sharp type schemes can also be inferred for recursive functions:

length		
`<null>`	$\to$	`fpi-zero`
`<cons>`	$\to$	`<number>`

In comparison, the one-line type scheme inferred in ML for an equivalent function is:

$$length : \forall\alpha.(list\ \alpha) \to integer$$

7. Special Inferences

We deviate from the unification process described so far when certain special functions are called and specific inference techniques must be applied:

equality predicates: The equality of values implies the equality of types, but the knowledge that two values are not equal does not allow anything to be inferred about the types of the values. This fact can not be expressed in type schemes for the standard equality predicates. Control flow inferencing is extended when standard equality predicates (e.g. *eq*, *neq*) occur to take advantage of these dependencies.

slot reader/writer: Access to a structure component uses a generic function, but the combination of the class of the structure and the name of the accessor identifies the slot concerned. In consequence, the type of the contents can be determined.

function arguments: In general, function types are not handled, but standard functions (like *apply*, *funcall*, *map*) are treated specially; argument and result types can easily be unified because the semantics of these functions are known.

8. Conclusions

A practical approach to type inference has been presented. The approach is realized in a module and application compiler for EuLisp and is based on unification over generic type schemes, which extend the expressiveness of common one-line type schemes. Using generic type schemes and through the use of a refined type lattice with singleton types we are able to retain and deduce more information about defined functions and constants.

The module structure of EuLisp aids greatly in improving both the efficiency of the type inferencing, and the detail of the information thus gained, particularly in the case of unexported functions by global inferencing.

References

1. Baker, H. G. The Nimble type inferencer for Common Lisp-84. (April 1990). Pre-puplication version.

2. Beer, R. D. Preliminary report on a practical type inference system for Common Lisp. *Lisp Pointers*, 1, 4 (1987) 5–11.

3. Bretthauer, H., Christaller, Th., Friedrich, H., Goerigk, W., Heicking, W., Hoffmann, U., Hovekamp, D., Knutzen, H., Kopp, J., Kriegel, E. U., Mohr, I., Rosenmüller, R., and Simon, F. Das Verbundvorhaben APPLY: Ein modernes und bedarfsgerechtes Lisp. *KI*, 2 (June 1992) 50–54.

4. Harper, R. *Introduction to Standard ML.* Report ECS-LFCS-86-14, University of Edinburgh (November 1986). Laboratory for Foundations of Computer Science.

5. Henglein, F. Global tagging optimization by type inference. In *Symposium on Lisp and Functional Programming*, ACM (1992) 205–215.

6. Jones, N. D. and Muchnick, S. Binding time optimization in programming languages. In *Third Symposium on Principles of Programming Languages* (1976) 77–94.

7. Kaplan, M. A. and Ullman, J. D. A scheme for the automatic inference of variable types. *Journal of the ACM*, 27, 1 (January 1980) 128–145.

8. Ma, K.-L. and Kessler, R. R. TICL—A type inference system for Common Lisp. *Software—Practice and Experience*, 20, 6 (June 1990) 593–623.

9. Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, , 17 (1978) 348–375.

10. Milner, R. A proposal for standard ML. In *ACM Symposium on Lisp and Functional Programming* (1984) 184–197.

11. Padget, J., Nuyens, G., and Bretthauer, H. (eds.). An overview of EuLisp. (1993). In this issue.

12. Robinson, J. A. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12, 1 (1965) 23–41.

13. Steenkiste, P. and Hennessy, J. Tags and type checking in Lisp: Hardware and software approaches. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (October 1987).

A. Formal Type Schemes of some Standard Functions

The formal type schemes of *integerp*, *cons*, *car*, *cdr* and + can be defined as follows:

integerp		
`<integer>`	$\rightarrow$	$\overline{\texttt{<null>}}$
$\overline{\texttt{<integer>}}$	$\rightarrow$	`<null>`

cons				
`<object>`	$\times$	$\overline{\texttt{mono-list} \sqcup \texttt{<null>}}$	$\rightarrow$	`poly-list`
$\overline{\texttt{<symbol>} \sqcup \texttt{<fpi>}}$	$\times$	`<null>`	$\rightarrow$	`poly-list`
$\overline{\texttt{<fpi>}}$	$\times$	`fpi-list`	$\rightarrow$	`poly-list`
$\overline{\texttt{<symbol>}}$	$\times$	`sy-list`	$\rightarrow$	`poly-list`
`<fpi>`	$\times$	`<null>` $\sqcup$ `fpi-list`	$\rightarrow$	`fpi-list`
`<symbol>`	$\times$	`<null>` $\sqcup$ `sy-list`	$\rightarrow$	`sy-list`

We abbreviate the standard type name `<fixed-precision-integer>` to `<fpi>` in the following schemes.

car		
`poly-list`	$\rightarrow$	`<object>`
`fpi-list`	$\rightarrow$	`<fixed-precision-integer>`
`sy-list`	$\rightarrow$	`<symbol>`

cdr		
`poly-list`	$\rightarrow$	`<object>`
$\alpha^{\texttt{mono-list}}$	$\rightarrow$	`<null>` $\sqcup$ $\alpha^{\texttt{mono-list}}$

$+$		
`poly-list`	$\rightarrow$	`<number>`
`fpi-list`	$\rightarrow$	`<fpi>`