Global Tagging Optimization by Type Inference*

Fritz Henglein
DIKU
University of Copenhagen
Universitetsparken 1
2100 Copenhagen Ø
Denmark
Internet: henglein@diku.dk

Abstract

Tag handling accounts for a substantial amount of execution cost in latently typed languages such as Common LISP and Scheme, especially on architectures that provide no special hardware support.

We present a tagging optimization algorithm based on type inference that is

global: it traces tag information across procedure boundaries, not only within procedures;

efficient: it runs asymptotically in almost-linear time with excellent practical run-time behavior (e.g. 5,000 line Scheme programs are processed in a matter of seconds);

useful: it eliminates at compile-time between 60 and 95% of tag handling operations in nonnumerical Scheme code (based on preliminary data);

structural: it traces tag information in higher order (procedure) values and especially in structured (e.g. list) values, where reportedly 80% of tag handling operations take place;

well-founded: it is based on a formal static typing discpline with a special type `Dynamic` that has a robust and semantically sound "minimal typing" property;

implementation-independent: no tag implementation technology is presupposed; the results are displayed as an explicitly typed source program and can be interfaced with compiler backends of statically typed languages such as Standard ML;

user-friendly: no annotations by the programmer are necessary; it operates on the program source, provides useful type information to a programmer in the spirit of ML's type system, and makes all tag handling operations necessary at run-time explicit (and thus shows which ones can be eliminated without endangering correctness of program execution).

This agenda is accomplished by:

- maintaining and tracing only a minimum of information — no sets of abstract closures or cons points *etc.* that may reach a program point are kept, only their collective tagging information; no repeated analysis of program points is performed;
- scheduling processing steps such that each one contributes to the final result — no idle or partially idle traversals of the syntax tree are performed; instead all relevant constraints are extracted in a single pass over the syntax tree;
- using theoretically and practically very efficient data structures; in particular, the union/find data structure is used to maintain and solve the extracted constraints.

This improves and complements previous work on tagging optimization in several respects.

- In the LISP compiler for S1 [BGS82], in Orbit [KKR*86], and in Screme [VP89,Ple91] tagging optimization (representation analysis) is typically performed for atomic types (numbers), based on local control flow information. Our analysis is global and based on abstract data flow information.
- In TICL [MK90] type analysis of Common LISP programs relies on costly repeated analysis and programmer type declarations.
- Shivers [Shi91a] similarly uses potentially expensive and complicated data flow reanalysis for type recovery in Scheme and relies to some degree on programmer type declarations; his analysis works on the CPS transform of a Scheme program and as such the results are not presentable to the user/programmer.

The main practical contribution of our tagging optimization algorithm is likely to be its combination of execution efficiency and ability to eliminate tag handling operations in structured data, especially in lists: Steenkiste and Hennessy report that 80% of all dynamic type checking operations are due to list operations, most of which are statically eliminated by our type inference algorithm. The computed information can also be used for unboxing and closure allocation (reference escape) analysis, although this is not pursued in this paper.

1 Introduction

LISP and its modern-day incarnations such as Common LISP [Ste84] and Scheme [Dyb87,Sch91] are latently typed languages. This means that data values carry specific *(type) tags* at run-time that identify the type of the value. Such a tag is chiefly used to type check the legality of operations executed at run-time; e.g., the application of the integer

*This research has been supported by Esprit BRA 3124, Semantique.

value 5 to the empty list () is illegal.[1] This is in contrast to statically typed languages such as Pascal and Standard ML where this check is performed at compile-time and any program containing a potentially type incorrect operation is categorically rejected as a whole. The practical trade-offs between latently typed and statically typed languages are well-known: flexibility and conciseness versus safety and efficiency of execution.

Execution efficiency of statically typed languages is typically better than that of latently typed languages for several reasons. First, static typing (usually) guarantees that all operations performed are legal, and consequently no run-time type checks need to be executed. Second, since tags and tag checking operations are unnecessary, less space is used for data and code, resulting in better data and code density. Third, compile-time type information can be used for type-specific storage management and instruction selection. Steenkiste and Hennessy [SH87,Ste91] report that a RISC implementation of Portable Standard LISP spends 9-25% of execution time on tag handling depending on the degree to which run-time type checking is actually performed. This figure does not even account for the more indirect effects on execution efficiency of reasons two and three above.

Static type information is also useful as a form of user-readable program documentation: explicit typed variable declarations (as in Pascal) specify programmer intent and are checked at compile-time without run-time penalty; inferred types (as in Standard ML [MTH90]) provide helpful information, both to the writer and other readers of a program, about fundamental properties of individual program parts.

In this paper we present a typing discipline and an efficient type inference algorithm that bring the dual benefit of execution efficiency and program documentation to latently typed languages without restricting the language in any way or requiring the programmer to provide explicit type information. The type inference algorithm eliminates at compile time most, but typically not all, tag handling operations.

We use Scheme as the language of discourse, but we wish to emphasize that the principles of tagging optimization based on "dynamic" type inference as expounded here are also applicable, with some additional complications, to other languages such as Common LISP, APL or SETL. Founded in the dynamically typed λ-calculus this tagging optimization technology is very robust, flexible and amenable to variations suitable to different languages and implementation technologies. For the same reason, however, it does *not* handle nor even address optimization issues that are "non-structural" in nature, such as the in practice very important optimization of number representations in Scheme.

In Section 2 we give an example of tagging in Scheme. We then describe the dynamically typed λ-calculus (Section 3), which constitutes the theoretical foundation and the core language for global tagging optimization by type inference. In Section 4 we show how the specific features of Scheme not addressed in the framework of the dynamically typed λ-calculus are handled. These are, amongst others, the initial top-level environment with its polymorphic primitive operations, side-effects, call-with-current-continuation, dynamic binding of top-level defined variables, garbage collection, type testing predicates and I/O. In Section 5 we describe the workings of the type inference algorithm that infers both type information and "minimal" tagging/untagging annotations. Section 6 presents data from a prototype implementation on the performance of global tagging optimization for Scheme. Both the performance of the tagging optimization algorithm and the quality of the optimizations are reported. Finally, Section 7 surveys related work and Section 8 concludes with an outlook on future work.

[1] Tags also serve the dual purpose of providing necessary information to a garbage collector. In fact in "naked" execution of programs, in which type checking operations are simply not performed, this may be their primary purpose!

```
(define lookup
  (lambda (key env)
    (if (equal? key (car (car env)))
        (cdr (car env))
        (lookup key (cdr env)))))

(define env-0
  (cons (cons 'x 5)
        (cons (cons 'id (lambda (x) x))
              '())))

(define env-1
  (cons (cons 'y #t)
        (cons (cons 'x (lambda (n) (+ n 1)))
              '())))

(+ (lookup 'x env-0) 8)

((lookup 'id env-0) 13)

(map (lambda (e) (lookup 'x e))
     (cons env-0 (cons env-1 '())))
```

Figure 1: Scheme source code for environment management

2 Tagging in Scheme: An Example

Consider the Scheme code in Figure 1. This code might be part of the environment management for a simple interpreter. The top-level variables `env-0` and `env-1` contain a list of pairs representing environments in which symbols are bound to values; e.g., the symbol `'x` is bound to the number `5` in `env-0` and to the (number-theoretic) successor function in `env-1`; the symbol `'id` is bound to the identity function in `env-0`. (The definitions of both `env-0` and `env-1` are written with `cons` operations instead of using quasi-quotation or the `list` operation to permit explicit indication of tag handling operations below.) The procedure `lookup` returns the value associated with an identifier. It assumes that such a value always exists. So, the procedure call `(+ (lookup 'x env-0) 8)` returns `13`, and so does `((lookup 'id env-0) 13)`. In the procedure call `(map (lambda (e) (lookup 'x e)) (cons env-0 (cons env-1 '())))` the function `(lambda (e) (lookup 'x e))`, which looks up the value bound to `'x` in its argument environment, is applied to both `env-0` and `env-1` and the results are returned as a list; viz., the list `(5 <successor-procedure>)`.

In Scheme, as in other latently-typed languages, values are tagged when they are *created*, and they are checked and untagged when they are *destructed (used)*. For example, any constant occurrence is a value creation point; so are cons- (for pairs) and lambda-expressions (for procedures). Procedure calls (for procedures) and `car` and `cdr` applications (for pairs) are destruction points; an addition operation de-

```
(define lookup
  [!PROC2
   (lambda ([key: Dynamic] [env: Dynamic])
     (if (equal? key (car [?PAIR (car [?PAIR env])]))
         (cdr [?PAIR (car [?PAIR env])])
         ([?PROC2 lookup] key (cdr [?PAIR env]))))])

(define env-0
  [!PAIR (cons [!PAIR (cons [!SYMBOL 'x] [!NUMBER 5])]
               [!PAIR (cons [!PAIR (cons [!SYMBOL 'id] [!PROC1 (lambda ([x: Dynamic]) x)])]
                            [!NULL '()])])])


(define env-1
  [!PAIR (cons [!PAIR (cons [!SYMBOL 'y] [!BOOLEAN #t])]
               [!PAIR (cons [!PAIR (cons [!SYMBOL 'x]
                                         [!PROC1 (lambda ([n: Dynamic])
                                           [!NUMBER (+ [?NUMBER n] [?NUMBER [!NUMBER 1]])])])]
                            [!NULL '()])])])

[!NUMBER (+ [?NUMBER ([?PROC2 lookup] [!SYMBOL 'x] env-0)] [?NUMBER [!NUMBER 8]])]

([?PROC1 ([?PROC2 lookup] [!SYMBOL 'id] env-0)] [!NUMBER 13])

(map [?PROC1 [!PROC1 (lambda ([e: Dynamic]) ([?PROC2 lookup] [!SYMBOL 'x] e))]]
     [!PAIR (cons env-0 [!PAIR (cons env-1 [!NULL '()])])])
```

Figure 2: Scheme code with explicit tag handling operations

structs its arguments and creates its result. Note that passing a value to a user-defined procedure or returning it from such a procedure are neither creation nor destruction points for the value.

If we make the tagging and untagging operations explicit in the Scheme code of Figure 1 then we get the annotated code displayed in Figure 2. Here we write `!T` for a tagging operation that tags its argument with type tag "T", and `?T` for the corresponding untagging operation that checks whether the tag of its argument is "T" and strips the tag if successful. For example, `!PROC1` is the tagging operation for "procedure with exactly one argument" and `?PROC2` is the untagging operation checking its argument for being a "procedure with exactly two arguments".

The explicit annotations reveal how much tagging and untagging is actually performed in a (naive) implementation. They also show places for opportunistic optimization. For example, the sequence of first tagging and then immediately untagging the constant number 8 in `[?NUMBER [!NUMBER 8]]` can be simplified to the untagged value 8; similarly, the sequence `[?NUMBER [!NUMBER 1]]` can be simplified to 1. Such local (intraprocedural) optimization for atomic types, notably numbers, is performed in several optimizing Scheme compilers; e.g., the S1 LISP compiler [BGS82] and Orbit [KKR*86]. But no currently existing compiler seems to attempt tagging optimization for structured or procedure values or to perform global (interprocedural) tagging optimization – and this even though Steenkiste and Hennessy [SH87] report that 80% of all dynamically executed type checking operations are due to list operations!

Our type inference algorithm is global, and it analyzes structured data such as lists and procedures. Furthermore it executes asymptotically in almost-linear time with a small constant factor and excellent run-time performance in practice (see Section 6). It manages to infer that lookup is only called with symbols in its first argument and with lists of pairs as its second argument where the first component of each pair is also a symbol. Consequently all tag handling operations in lookup can be eliminated and the dynamically overloaded `equal?` predicate can be determined to be an equality comparison on symbols, which may then be inlined in the compiled code. Yet, it also discovers that the second components of pairs in environments need to be tagged at their creation points and untagged when they are used. Specifically, it infers that only the explicit tag handling operations in Figure 3 are necessary. We use the types $\mathtt{NPair}(t, u) = t * u + \mathtt{Null}$ and $\mathtt{List}(t) = \mu t.\mathtt{NPair}(t, u)$ in the annotated code; that is, $\mathtt{NPair}(t, u)$ is the type of (untagged) pairs plus the empty list, `'()`, and $\mathtt{List}(t)$ is the type of (untagged!) lists of element type t.

3 A core language: the dynamically typed λ-calculus

We start with a minimal, but paradigmatic programming language: the *dynamically typed λ-calculus.* This is an extension of the (statically) typed λ-calculus with a distinguished type constant `Dynamic` representing tagged objects and coercions between tagged and untagged values. For every primitive type and type constructor we have exactly one *(type) tag.* Untagged values are transformed into tagged values, and vice versa, by *tag handling operations (coercions)*: a *tagging* operation takes as input an untagged value, adds the corresponding tag and returns the tagged value; an *untagging* operation takes as input a tagged value, checks the value for a specific tag and, if successful, removes the tag and returns the resulting untagged value; if unsuccessful, it aborts the program.

Some of the basic theoretical and type inference properties of dynamic typed λ-calculus are investigated in [Hen92]. Note that for every primitive type and type constructor there is only one tag. This is in contrast to dy-

```
(define lookup
  (lambda ([key: Symbol] [env: List(Symbol, Dynamic)])
    (if (equal? key (car (car env)))
        (cdr (car env))
        (lookup key (cdr env)))))

(define env-0
  (cons (cons 'x [!NUMBER 5])
        (cons (cons 'id [!PROC1 (lambda ([x: Dynamic]) x)])
              '())))

(define env-1
  (cons (cons 'y [!BOOLEAN #t])
        (cons (cons 'x [!PROC1 (lambda ([n: Dynamic]) [!NUMBER (+ [?NUMBER n] 1)])])
              '())))

(+ [?NUMBER (lookup 'x env-0)] 8)

([?PROC1 (lookup 'id env-0)] [!NUMBER 13])

(map (lambda ([e: List(Symbol, Dynamic)]) (lookup 'x e))
     (cons env-0 (cons env-1 '())))
```

Figure 3: Result of global tagging optimization

namic type systems that permit tagging with complete *types* [Myc84,ACPP91,LW91].

Tag handling operations are implicit in latently typed languages. We make them explicit in order to be able to eliminate (some of) them at compile-time – and to be sure that they are really eliminated when they do not occur in the final program. In Section 4 we discuss how type and coercion inference for the dynamically typed λ-calculus can be extended to additional types as well as the imperative and dynamic aspects of Scheme. In Section 5 we show how a "minimum" number of such tag handling operations necessary for static type correctness can be automatically inferred in almost-linear time for a given program. Empirical results suggest that 60-95% of all tag handling operations in non-numerical programs, both in terms of syntactic occurrences and execution frequency, can be eliminated in this fashion.

In the type inference system considered in [Hen92] arbitary coercions may be inferred for arbitrary expressions; even *induced* coercions as in $[(\texttt{?FUNC} \rightarrow \texttt{!FUNC})f]$, which denotes the expression $\lambda x : \texttt{Dynamic}.[\texttt{!FUNC}\,(f[\texttt{?FUNC}\,x])]$. For tagging optimization we are only interested in completions of untyped λ-terms in which only primitive (no induced) coercions occur, and those may only occur at creation and destruction points, if at all. (See also Section 8 for a practically useful relaxation of this requirement.) Specifically, for the dynamically typed λ-calculus this means that: a `!FUNC` coercion may only occur applied to a lambda abstraction; a `?FUNC` coercion may only occur applied to the function in a function application; a `!BOOL` coercion may only occur applied to the constants `#t` and `#f`; and `?BOOL` may only occur applied to the test clause in a conditional; analogously for other types. (In the terminology of [Hen92] this corresponds to the class of completions C_{pf}.)

The resulting typing rules for the dynamic λ-calculus with Booleans are given in Figure 4. Note that we use notational conventions familiar from the literature on type inference systems, but adopt Scheme-like notation for the program syntax. The type expressions in this language are generated by the production

$$\tau ::= \alpha \mid \texttt{Dynamic} \mid \mathit{bool} \mid \tau' \rightarrow \tau'' \mid \mu\alpha.\tau'.$$

Here α ranges over a set of type variables and $\mu\alpha.\tau'$ denotes a recursive type for which the equality $\alpha = \tau'$ holds. Other types and type constructors can be treated analogously. The special coercion `NOOP` is an *improper* coercion. It models the case when no tag handling operation is performed at all. Thus we may simply write (ee') instead of $([\texttt{NOOP}\ e]e')$, *etc.*. We call the other coercions *proper*.

A dynamically typed λ-term is a term e such that $A \vdash e : \tau$ is derivable in this type inference system for some A, τ. Its *erasure* is the underlying untyped λ-term resulting from erasing all type information and all occurrences of coercions from e. Conversely, a dynamically typed λ-term $\bar{e}$ is a *completion* of an untyped λ-term e if e is the erasure of $\bar{e}$.

An untyped λ-term may have several different completions. For example, the untyped λ-term

$$e_0 = ((((\texttt{lambda}\,(b)(\texttt{lambda}\,(x)(\texttt{lambda}\,(y)\ (\texttt{if}\ b\ x\ y))))\texttt{\#t})\texttt{\#t})\texttt{\#f})$$

has, amongst others, the completions displayed in Figure 5. The completion $\bar{e}_0^c$ is *canonical* in the sense that every subexpression of e_0 is coerced to a value of type `Dynamic` and every function expects a value of type `Dynamic` as input.

We say a completion $\bar{e}$ of an untyped λ-term e is *minimal* if for every subterm e' in e a proper coercion `C` is applied to e' in $\bar{e}$ if and only if `C` is applied to e' in *all* completions of e. The completion $\bar{e}_0^m$ is clearly minimal in this sense since it uses no proper coercions at all, and thus also optimal since all tag handling operations are eliminated.

4 Extending the type system to Scheme

The dynamically typed λ-calculus of Section 3 constitutes the core of a type inference system for Scheme. The *primitive* types for Scheme are `Null`, `Number`, `Boolean`, `Character`, `String`, `Symbol`. They represent *untagged* atomic values.

$$
\begin{array}{ll}
e_0 = & (((\,(\texttt{lambda}\,(b)(\texttt{lambda}\,(x)(\texttt{lambda}\,(y) \\
& \quad (\texttt{if}\ b\ x\ y)\ ))) \\
& \quad\quad \texttt{\#t})\ \texttt{\#t})\ \texttt{\#f}) \\
\\
\bar{e}_0^m = & (((\,(\texttt{lambda}\,(b:\texttt{Boolean})(\texttt{lambda}\,(x:\texttt{Boolean})(\texttt{lambda}\,(y:\texttt{Boolean}) \\
& \quad (\texttt{if}\ b\ x\ y)\ ))) \\
& \quad\quad \texttt{\#t})\ \texttt{\#t})\ \texttt{\#f}) : \texttt{Boolean} \\
\\
\bar{e}_0^c = & ([\texttt{?FUNC}\,([\texttt{?FUNC}\,([\texttt{?FUNC}\ \ [\texttt{!FUNC}\,(\texttt{lambda}\,(b:\texttt{Dynamic})[\texttt{!FUNC}\,(\texttt{lambda}\,(x:\texttt{Dynamic})[\texttt{!FUNC}\,(\texttt{lambda}\,(y:\texttt{Dynamic}) \\
& \quad (\texttt{if}\ [\texttt{?BOOL}\ b]\ x\ y))])])]] \\
& \quad\quad [\texttt{!BOOL}\,\texttt{\#t}])][\texttt{!BOOL}\,\texttt{\#t}])][\texttt{!BOOL}\,\texttt{\#f}]) : \texttt{Dynamic}
\end{array}
$$

Figure 5: Two possible completions of e_0

$$A \vdash \texttt{x} : \tau \qquad \text{if } \texttt{x} : \tau \text{ in } A$$

$$\frac{A, \texttt{x} : \tau \vdash e : \tau'}{A \vdash [\texttt{C}\,(\texttt{lambda}\,(\texttt{x} : \tau)\ e)] : \tau''} \quad (\texttt{C} : (\tau \to \tau') \leadsto \tau'')$$

$$\frac{\begin{array}{c} A \vdash e : \tau \\ A \vdash e' : \tau' \end{array}}{A \vdash ([\texttt{C}\,e]\ e') : \tau''} \quad (\texttt{C} : \tau \leadsto (\tau' \to \tau''))$$

$$A \vdash [\texttt{C}\ \texttt{\#t}] : \tau \qquad (\texttt{C} : \texttt{Boolean} \leadsto \tau)$$

$$A \vdash [\texttt{C}\ \texttt{\#f}] : \tau \qquad (\texttt{C} : \texttt{Boolean} \leadsto \tau)$$

$$\frac{\begin{array}{l} A \vdash e : \tau \\ A \vdash e' : \tau' \\ A \vdash e'' : \tau'' \end{array}}{A \vdash (\texttt{if}\ [\texttt{C}\ e]\ e'\ e'') : \tau'} \quad \left(\begin{array}{l} \tau' = \tau'' \\ \texttt{C} : \tau \leadsto \texttt{Boolean} \end{array}\right)$$

$$\texttt{!FUNC} : (\texttt{Dynamic} \to \texttt{Dynamic}) \leadsto \texttt{Dynamic}$$

$$\texttt{?FUNC} : \texttt{Dynamic} \leadsto (\texttt{Dynamic} \to \texttt{Dynamic})$$

$$\texttt{!BOOL} : \texttt{Boolean} \leadsto \texttt{Dynamic}$$

$$\texttt{?BOOL} : \texttt{Dynamic} \leadsto \texttt{Boolean}$$

$$\texttt{NOOP} : \tau \leadsto \tau$$

Figure 4: Typing rules for the dynamically typed λ-calculus

The *compound* types are $\tau_1 * \tau_2, \tau_1 \to \tau_2, \texttt{Vector}\ \tau$ representing untagged pairs, procedures and vectors, respectively. Other types such as input and output ports, multiadic and variadic procedure types (see below) may be added as needed. Finally, we add simple regular recursive types of the form $\mu t.\tau[t]$. An initial type environment contains typings for the primitive operations on these types.

In this section we show how the dynamically typed λ-calculus can be extended to address the features of Scheme that require special attention. These are:

1. multiadic and variadic procedure types;
2. primitive operations and polymorphism;
3. side effects, mutable objects, and continuations;
4. dynamically scoped top-level bindings;
5. garbage collection;
6. I/O and type testing routines.

4.1 Multiadic and variadic procedure types

Scheme has multiadic and variadic procedures. We write $[\alpha_1 \ldots \alpha_k] \to \beta$ for the type of a k-ary procedure, and $[\alpha_1 \ldots \alpha_k]\alpha \to \beta$ for the type of a variadic procedure with at least k-arguments (writing $\alpha \to \beta$ if k is 0). That is, $\to$ actually represents a whole *class* of type constructors.

In the implementation of dynamic type inference we do not need multiadic and variadic procedure types. We can treat a Scheme procedure call $(p\ e_1\ e_2 \ldots e_k)$ as a unary procedure call to p with argument

$$(\texttt{cons}\ e_1(\texttt{cons}\ e_2(\ldots(\texttt{cons}\ e_k\ '())\ldots)))$$

and a lambda expression $(\texttt{lambda}\ (x_1 \ldots x_l)e)$ correspondingly as a "pattern-matching" definition of a unary procedure

$$(\texttt{lambda}\,(\texttt{cons}\ x_1(\texttt{cons}\ x_2(\ldots(\texttt{cons}\ x_l\ '())\ldots)))e)$$

. A variadic procedure definition $(\texttt{lambda}\ (x_1 \ldots x_k\ .\ y)\ e)$ is treated as

$$(\texttt{lambda}\,(\texttt{cons}\ x_1(\texttt{cons}\ x_2(\ldots(\texttt{cons}\ x_l\ y)\ldots)))e)$$

, and $(\texttt{lambda}\ xe)$ corresponds to itself. At first sight it may seem surprising that this simple encoding should be correct since list-valued objects are clearly different from formal and actual argument lists in Scheme programs. The syntax of Scheme guarantees, however, that all procedure calls must have argument lists (not a single list-valued argument!) and thus actual argument lists are matched correctly with formal argument lists in the type inference system. While it would be possible to handle multiadic and variadic procedure types directly in the implementation, this treatment as unary procedures is simple, especially for variadic procedures. For example, the operation `list` can be given the type scheme

$$\texttt{list} : \forall\alpha.\alpha \to \alpha.$$

4.2 Primitive operations and polymorphism

Primitive operations for Scheme's data types are frequently polymorphic; that is, `cons` applied to two numbers returns a pair of numbers whereas applied to two symbols it returns a pair of symbols. Any attempt at successful tagging optimization must recognize and take into account the fundamental polymorphic properties of the primitive operations. Our type system presented above is *monomorphic* in the sense that every *user-defined* object is required to have a single non-polymorphic type. The initial type environment, however, contains *type schemes* for the primitive operations. For example, using the notation introduced in 4.1 it contains

$$\begin{array}{rcl} \texttt{car} & : & \forall\alpha\beta.[\alpha * \beta] \to \alpha \\ \texttt{cdr} & : & \forall\alpha\beta.[\alpha * \beta] \to \beta \\ \texttt{cons} & : & \forall\alpha\beta.[\alpha\ \beta] \to (\alpha * \beta) \end{array}$$

for the basic list operations. These types model the functionality of the operations without any tag handling operations. Every occurrence of a primitive operation in a program receives a fresh copy of this type, with suitable coercion parameters to account for possible tagging/untagging operations on its arguments, its result and itself. For `car`, this is

$$\texttt{car}[c : ([\alpha * \beta] \to \alpha) \rightsquigarrow \gamma] : \gamma$$

where α, β, γ are fresh type variables.

Application of `car` to a pair of numbers where the result is added to another number results in the coercion parameterization `car[NOOP]` in the program's minimal completion whereas passing `car` to a procedure that expects a tagged value results in `car[!PROC2]`. The result of the parameterization in the first case is an untagged procedure of type $[\texttt{Number} * \texttt{Number}] \to \texttt{Number}$ and in the second case a tagged object of type `Dynamic`. Note that the parameterizations can be compiled to specialized code in accordance with the actual coercion parameters.

4.3 Side effects, mutable objects and continuations

The well-known complications and dangers of polymorphic type inference in the presence of pointers and side-effecting operations [Tof90] occur only when a naive polymorphic type generalization rule for *user-defined* objects is used. The same comment applies to the treatment of continuations [HL].

Side effecting operations and mutable objects require no special attention since our type inference system is *monomorphic*: every user-defined object is required to have a single type, not a type scheme. This may make tagging elimination more conservative than necessary, but not unsound. Our initial environment contains the following type schemes:

$$\begin{array}{rcl} \texttt{set!} & : & \forall\alpha\beta.[\alpha\ \alpha] \to \beta \\ \texttt{set-car!} & : & \forall\alpha\beta\gamma.[(\alpha * \beta)\ \alpha] \to \gamma \\ \texttt{set-cdr!} & : & \forall\alpha\beta\gamma.[(\alpha * \beta)\ \beta] \to \gamma \\ \texttt{call/cc} & : & \forall\alpha\beta.[[[\alpha] \to \beta] \to \alpha] \to \alpha \end{array}.$$

Individual occurrences of these operations in a program are treated as indicated in the previous subsection. For example, an instance of `call/cc` would be parameterized by three coercions:

$$\begin{array}{ll} \texttt{call/cc} & [c_1 : ([\alpha] \to \beta) \rightsquigarrow \gamma, \\ & c_2 : ([\gamma] \to \alpha) \rightsquigarrow \gamma', \\ & c_3 : ([\gamma'] \to \alpha) \rightsquigarrow \gamma''] : \gamma''. \end{array}$$

4.4 Dynamically scoped top-level bindings

Top-level bound variables are dynamically scoped in Scheme. This provides a flexible programming environment, but makes programming potentially hazardous and compilation to efficient code very difficult. Our `lookup` definition of Figure 1 is a case in point: after `(define lookup2 lookup)` `(define lookup ...)` the procedure bound to `lookup2` is not the one originally bound to `lookup`. Our dynamic type inference algorithm takes a conservative approach to rebindings of top-level variables: the object bound to a particular variable before and after a rebinding must have the same type. This may result in a more conservative completion than strictly necessary, but not in an unsound one. It has the advantage that constraint normalization (see Section 5) can be performed on-line, processing one top-level Scheme command at a time. Clearly, the benefits of global tagging optimization are greatest if we are allowed to "freeze" the bindings of a Scheme program when compiling it.

4.5 Garbage collection

Even though it may be possible to eliminate some tagging operations for typing purposes the garbage collector may still require explicit tags on data objects. The main benefits to be gained from tagging optimization, however, are the elimination of untagging operations since the cumulative cost of tagging operations appears to be only marginal [SH87]. Dynamic type inference offers, however, the prospect of bringing the newly emerging technology of tag-free garbage collection for statically typed programming languages to the realm of run-time typed languages [Gol91].

4.6 I/O and type testing routines

We give each of the type testing routines `number?, symbol?` *etc.* the type $[\texttt{Dynamic}] \to \texttt{Boolean}$. It is also possible (and gives potentially better results) to give them the type scheme $\forall\alpha.[\alpha] \to \texttt{Boolean}$. In this case specialized code can be generated for any nonvariable instantiation of α.

Similarly, the standard I/O routines `read` and `write` have type/type scheme

$$\begin{array}{rcl} \texttt{read} & : & [\texttt{Null}] \to \texttt{Dynamic} \\ \texttt{write} & : & \forall\alpha.[\texttt{Dynamic}] \to \alpha \end{array}$$

Again, we may give `write` the type scheme $\forall\alpha\beta.[\beta] \to \alpha$, which corresponds to a typing for a family of specialized output routines.

5 Dynamic type inference algorithm

Our type inference algorithm takes as input an (untyped) Scheme program and computes its minimal completion (minimal in the sense of Section 3). In particular, it computes:

- for every locally bound variable a simply typed variable declaration;
- a top-level environment of type bindings (binding top-level variables to simple type expressions);
- for every data construction point in the program whether or not a proper tagging operation is necessary;
- for every data destruction point whether or not a proper untagging operation is necessary.

The algorithm proceeds in four conceptual phases:[2]

Parsing: The input program is parsed and an abstract syntax tree is constructed. Variable occurrences are classified into locally-bound and top-level occurrences; locally-bound variable occurrences share the corresponding binding occurrence. We associate a unique *type variable* with every node in the abstract syntax tree and every top-level variable; and a unique *coercion variable* with every node to which a coercion may *potentially* be applied; *i.e.*, at the data creation and destruction points.

Constraint generation: Based on the side conditions of the typing rules constraints on the type variables and coercion variables are generated whose solutions characterize the set of *all* completions of the input program.

Constraint normalization: The set of type and coercion constraints is *normalized* to an equivalent set of constraints using an efficient union/find based algorithm.

Minimal completion annotation and pretty printing: The normalized set of constraints is solved for its *minimal* completion, which is then output as type declarations of locally bound and top-level variables and as explicit tag handling operations inserted into the source program.

The thus completed program is guaranteed to be statically type correct; in particular, only those type handling operations explicitly in the annotated program need to be implemented.

We shall describe the first two phases by example. Constraint normalization, being the heart of the algorithm, is presented in more detail after that.

5.1 Parsing and constraint generation

Consider the definition of `lookup` in Figure 1. After parsing and annotation with type variables and coercion variables we obtain the abstract syntax tree shown in Figure 6. (The type variables associated with subexpressions are omitted for reasons of readability. Below they are identified by their subscripts.)

Every completion of `lookup` must have the form of the annotated syntax tree. A completion is valid if and only if the constraints listed as side conditions in the typing rules of Figure 4 are satisfied. Some of these constraints for the definition of `lookup` are listed in the left colum of Figure 7.

Let us define $\tau \leq \tau'$ if $C : \tau \rightsquigarrow \tau'$ where C is some *tagging* operation or the improper coercion `NOOP` . Since a coercion is completely identified by its type signature, it is easy to see that the constraints listed in the right column of Figure 7 are equivalent to the constraints in the left column. Furthermore, all the constraints generated in this fashion have one of the two forms

- $f^{(k)}(\alpha_1, \ldots \alpha_k) \leq \alpha$,
- $\alpha = \alpha'$

where $f^{(k)}$ is a k-ary type constructor (a primitive type, if $k = 0$, but not `Dynamic`), and $\alpha, \ldots$ range over type variables and `Dynamic`. A *solution* of a constraint system is a substitution of types for the type variables such that all constraints are satisfied.

[2] The first three phases can be executed in a single pass over the input program.

5.2 Constraint normalization and minimal solutions

The constraints generated from a program characterize all its possible (valid) completions. Constraint normalization is the process of rewriting these constraints into an equivalent normal form from which the minimal completion can be easily constructed. This is analogous to the process of unification in ML-like languages, with the difference that coercions can be inserted as required instead of reporting a complete type failure. The normalization can be specified by a multiset rewriting system, given in Figure 8.

It is easy to check that the set of solutions is preserved by rewriting steps. Due to the inequality constraint rules and the availability of recursive types all inequality constraint rules remaining in the normalized constraint system can be solved equationally; i.e., using the most general (circular) unifier. Since an inequality solved equationally gives a `NOOP` solution to the corresponding coercion constraint this gives the desired minimal completion of the input program. When applying this algorithm to the program in Figure 1 the resulting minimal completion for our example program is the one presented Figure 3.

In [Hen91] it is shown that, using an instrumented unification closure algorithm and the union/find data structure with ranked union and path compression [Tar83], constraint normalization can be implemented in almost linear time (in terms of the size of the input program). This bound even applies in the case when coercions are permitted anywhere, not just at data creation and destruction points, and constraints of the form $\alpha \leq \beta$ must be considered. Furthermore, this algorithm exhibits excellent practical run-time behavior as evidenced in Section 6.

6 Status and experimental results

We have implemented a rudimentary version of dynamic type inference for Scheme based on an implementation of the union-find data structure with path compression and union-by-rank (see, *e.g.*, [Tar83, Section 2]). The preliminary results are very encouraging: the current algorithm, which still misses ample opportunities for eliminating tag handling operations, eliminates typically more than 60% of all occurrences of tag handling operations in nonnumerical code. See Figure 1 for some results of tagging optimization by dynamic type inference using a Scheme-based prototype implementation.

The first column in the table gives the name of the Scheme program analyzed, the second its size in kilobytes, the third the number of constraints generated, the fourth the execution time of constraint normalization *without* garbage collection, the fifth an averaged garbage collection time for constraint normalization, the sixth the percentage of tagging operations remaining in the program, the seventh the percentage of tagging operations eliminated, the eighth the percentage of untagging operations remaining, and the ninth the percentage of untagging operations eliminated. Percentages are rounded to the nearest 5%. Garbage collection time is listed separately since it tends to vary widely between different test runs whereas execution time without it has shown itself to be more consistent. At present the running time of constraint normalization is completely dominated by parsing and pretty-printing. This is partly due to the fact that the parser is coded without regard to efficiency (in particular, it generates a lot of garbage) and partly to the core constraint normalization routines being coded rather care-

```
(define (lookup: t0)
  [C0 (lambda ((key: t1) (env: t2))
    (if [C1 ([C2 equal?[C3]] key ([C4 car[C5]] ([C6 car[C7]] env)))]
        ([C8 cdr[C9]] ([C10 car[C11]] env))
        ([C12 lookup] key ([C13 cdr[C14]] env))))])
```

Figure 6: Type and coercion annotated definition of `lookup`

$\mathtt{C0} : [t1\ t2] \rightarrow t_{(\mathtt{if}\ \ldots)} \rightsquigarrow t_{(\mathtt{lambda}\ \ldots)}$	$[t1\ t2] \rightarrow t_{(\mathtt{if}\ \ldots)} \leq t_{(\mathtt{lambda}\ \ldots)}$
$t0 = t_{(\mathtt{lambda}\ \ldots)}$	
$\mathtt{C1} : t_{(\mathtt{equal?}\ \ldots)} \rightsquigarrow \mathtt{Boolean}$	$\mathtt{Boolean} \leq t_{(\mathtt{equal?}\ \ldots)}$
$\mathtt{C2} : t_{\mathtt{equal?}} \rightsquigarrow [t1\ t_{(\mathtt{car}\ (\mathtt{car}\ \mathtt{env}))}] \rightarrow t_{(\mathtt{equal?}\ \ldots)}$	$[t1\ t_{(\mathtt{car}\ (\mathtt{car}\ \mathtt{env}))}] \rightarrow t_{(\mathtt{equal?}\ \ldots)} \leq t_{\mathtt{equal?}}$
...	...

Figure 7: Constraints for completions of `lookup`

fully. All in all the parser takes about five times as long as constraint normalization. This includes constraint generation, which accounts for roughly half this time. All tests have been performed using Chez Scheme, version 3.2, on a Sun SparcStation 2 under SunOS 4.1.1. The execution times are as reported by the `time` procedure.

Here is a short description of the analyzed programs (their authors are given in parentheses).

reg-int: A regular expression interpreter that accepts a regular expression and a string as input and checks whether the string is an element of the language generated by the regular expression (by Anders Bondorf, Torben Mogensen and Jesper Jørgensen).

meta: A meta interpreter that translates a denotational definition (possibly with lazy functions) into a Scheme interpreter (by Jesper Jørgensen) for the language.

specializer: The static execution and code generation component of the specializing partial evaluator Similix (version 4.0, by Anders Bondorf).

dyn-typ: The current implementation of dynamic type inference, including parsing and pretty printing, which accounts for 80% of the code (by the author).

cogen: A compiler generator generating compilers from interpreters written in Scheme (generated from the partial evaluator Similix by self-application, see [Bon91]).

cmp: A complete script compiler for the lazy functional programming language BAWL ("Bird-and-Wadler language") (by Jesper Jørgensen, generated from an interpreter using Similix, see [Jr92]).

Similar numbers as for textual elimination of tag handling operations appear to hold for the dynamic execution of tag handling operations. So far we have only collected preliminary run-time statistics when executing the dynamic type inference system itself. Here the figures are: of all potential tagging operations 60% of the executed ones are those of the tagging operations left in the program, 40% of those that are eliminated. For untagging operations this ratio is 45%/55% in favor of eliminated untagging operations.

Even more encouraging are the figures on the performance of the type inference algorithm itself, in particular the constraint normalization algorithm: it solves more than 2,000 constraints per second on a Sun SparcStation 2 *disregarding* garbage collection. Since a 100,000 line Scheme program or set of programs generates about 200,000 constraints, this would predict a running time of less than 2 minutes, which appears acceptable for an optimizing compiler, especially since the constraints can be generated and solved incrementally in a sequential pass over the source program. The actual running time is bound to be a lot worse due to disproportionately increased garbage collection overhead, swapping and loss of locality of reference, amongst other things.

7 Related work

The dynamic λ-calculus of Section 3 is reminiscent of a subtyping discipline, but it is critically different in that both tagging *and* untagging coercions are present. In this regard it is closely related to Thatte's notion of quasi-static typing [Tha90] where tagging and untagging are referred to as positive and negative coercions, respectively. (A subtyping discipline has only positive coercions.) In Thatte's language, however, the types of bound variables must be explicitly declared and it is thus not suitable for application in automatic tagging optimization. Gomard [Gom90] describes type inference for implicitly typed programs with no required type information at all. In his type system there are no untagging operations for first-order values, but instead tagged versions of base operations are used. As a consequence tagging may "spread" to every point reachable from a single tagging operation. Cartwright and Fagan [CF91] present a unification-based "soft" typing system as an extension of ML in which static typing failures are handled during unification by opportunistic insertion of run-time type checks. Even though their type inference algorithm collects very detailed type information it is not clear whether the result can be used for tagging optimization; instead it appears that all values are expected to be fully tagged at run-time.[3]

Collecting type information for latently typed languages is far from new. Type finding based on classical data flow analysis technology goes back to Tenenbaum [Ten74], Jones and Muchnick [JM76,JM82], and Kaplan and Ullman [KU80]; more recently it has been extended and adapted to SETL by Weiss [Wei86]; to Icon by Walker [Wal88]; to Common LISP by Ma and Kessler [MK90]; to Scheme by Shivers [Shi91a] (see below). Type information is also instrumental in optimizing the implementation of functional

[3] The article [CF91] contains some flaws and is currently under revision.

1. (Inequality constraint rules)
   (a) $C \cup \{f^{(k)}(\alpha_1 \ldots \alpha_k) \leq \gamma, f^{(k)}(\beta_1 \ldots \beta_k) \leq \gamma\} \Rightarrow C \cup \{f^{(k)}(\alpha_1 \ldots \alpha_k) \leq \gamma, \alpha_1 = \beta_1, \ldots, \alpha_k = \beta_k\}$
   (b) $C \cup \{f^{(k)}(\alpha_1 \ldots \alpha_k) \leq \gamma, g^{(l)}(\beta_1, \ldots, \beta_l) \leq \gamma\} \Rightarrow C \cup \{\alpha_1 = \texttt{Dynamic}, \ldots, \alpha_k = dyn, \beta_1 = \texttt{Dynamic}, \ldots, \beta_l = \texttt{Dynamic}, \gamma = \texttt{Dynamic}\}$ if $f^{(k)} \neq g^{(l)}$;
2. (Equational constraint rules)
   (a) $C \cup \{\alpha = \alpha\} \Rightarrow C$;
   (b) $C \cup \{\texttt{Dynamic} = \alpha\} \Rightarrow C \cup \{\alpha = \texttt{Dynamic}\}$ if α is a type variable;
   (c) $C \cup \{\alpha = \alpha'\} \Rightarrow C[\alpha'/\alpha] \cup \{\alpha = \alpha'\}$ if $\alpha \neq \alpha'$ and α is a type variable with at least one occurrence in C.

Figure 8: Constraint normalization

name	*size (kByte)*	*constraints*	*time (gc)*		*tags/no-tags*		*untags/no-untags*	
reg-int	0.7	31	0.0		0%	100%	0%	100%
meta	4.3	336	0.1		25%	75%	20%	80%
specializer	15.1	500	0.2		30%	70%	10%	90%
cogen	65.1	2,569	1.1	(1.2)	50%	50%	25%	75%
dyn-typ	66.2	2,685	1.0	(0.9)	35%	65%	35%	65%
cmp	232.6	8,946	4.3	(15.0)	15%	85%	35%	65%

Table 1: Statistics on performance of dynamic type inference prototype

languages [AM91] and object-oriented languages; e.g., in Smalltalk [JGZ88] and Self [CU90].

The application of type finding and type information to tagging optimization can be traced to (at least) the early work on SETL optimization [FSS75,SSS81], which is based on Tenenbaum's thesis referenced above. Intraprocedural representation analysis for local optimization of primitive operations (operating on atomic data) is described in Brooks, Gabriel and Steele's S1 compiler for (an extension of) Common LISP [BGS82] and mentioned in Vegdahl and Pleban's article [VP89] on the run-time system of their Screme compiler for Scheme. Ma and Kessler's type inference system for Common LISP, TICL [MK90], seeks to propagate programmer type declarations to other parts of Common LISP programs by forwards analysis, repeatedly analyzing procedure definitions in different call contexts. Their reanalysis is potentially costly (exponential time), and the combination of type information for a procedure from different call contexts appears to yield potentially unsound typings.

The most comprehensive study to date on type analysis in Scheme is Shivers' work [Shi88,Shi91a,Shi91b]. He extends a classical forward flow analysis to higher-order values by modeling procedures as abstract closures and using a collecting abstract interpretation to associate sets of such closures with every call point in a program; at these call points the type analysis is repeatedly executed and the type information updated until it stabilizes. The type information his method computes appears better in some respects than ours since he takes control flow information into account, which is ignored in our approach. The combination of repeated analysis with the need to compute abstract closure sets suggests that his method may be inherently too inefficient for medium- to large-sized programs, though, as the fastest algorithm known to us for closure analysis [Ses89] alone is very complicated and requires $\Theta(n^3)$ time.

It is worthwhile pointing out that Peterson [Pet89] addresses the orthogonal problem of minimizing the (dynamic) number of changes between the tagged and untagged representation of an object that is, in principle, tagged. It is conceivable that his techniques can be combined with dynamic type inference since they are based on local control flow information, which is at present ignored in our approach; however, in his model there are operations – notably the list operations – that are *required* to take tagged operands, which manifestly precludes tagging optimization of list operations.

8 Conclusion

Dynamic type inference provides a robust, implementation-independent, efficient way of eliminating tag handling operations in run-time typed languages. It facilitates the modular construction of compilers and other tools and bridges the gap between statically typed and run-time typed languages, both with regard to use and implementation. Much remains to be done to turn dynamic type inference into a practical technology, however. Some work that is underway in this direction and some possible future work is outlined below.

- Relaxing the rules on where tag handling operations may occur in a program may result in better completions. For example, in the type system described here tagging operations may only be applied at data creation points. This entails that, e.g., a number that is first used in arithmetic operations several times and eventually written to the output must be tagged at the creation point and repeatedly untagged by the arithmetic operations. It is clearly better to delay tagging until the number is passed to the output routine, which we assume requires a tagged input value. A type system permitting coercions potentially at any point still admits a very fast inference algorithm (see [Hen91] for its use in binding-time analysis). We are in the process of implementing this algorithm.

- A project to use dynamic type inference to build a translater from Scheme to the lambda intermediate language of the SML/NJ compiler has just begun at DIKU.
- The outlines of a polymorphic dynamic typing discipline with polymorphic objects parameterized by coercions are already in [Hen92], but more work is necessary. The possibility of parameterizing objects with coercions and specializing the code with respect to these may be relevant for the efficient compilation of top-level Scheme definitions.
- A form of strictness analysis, usually only used in languages with lazy evaluation, appears to be desirable to "push" as many coercions as possible from inside procedure definitions back to their arguments or forwards to their results and thus out of the definitions.
- Dynamic type inference based tagging optimization could be combined with flow control information and data flow analysis, as in [Shi91a], to achieve better results. For example, when making the type handling operations explicit in the intermediate code of a language avoiding repeated (un)tagging of the same object becomes an instance of the common subexpression elimination problem.

Acknowledgements

I would like to thank Olivier Danvy for his encouragments, his early and constructive criticisms on general and specific matters of this work, and particularly for pointing out the special features of Scheme semantics that needed to be addressed explicitly. Satish Thatte and Carsten Gomard have provided the original incentive and ongoing feedback to this work. Thanks also to Thomas Breuel for sharing his thoughts on compile time optimization of dynamically typed languages with me. Finally I would like to thank Bob Paige, Konstantin Läufer and Kees van Schaik for facilitating the completion of this paper and express my gratitude to the TOPPS group at DIKU for their usual support.

References

[ACPP91] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):237–268, April 1991. Presented at POPL '89.

[AM91] A. Aiken and B. Murphy. Static type inference in a dynamically typed language. In *Proc. 18th Annual ACM Symp. on Principles of Programming Languages, Orlando, Florida*, pages 279–290, ACM, ACM Press, Jan. 1991.

[BGS82] R. Brooks, R. Gabriel, and G. Steele. An optimizing compiler for lexically scoped LISP. In *Proc. SIGPLAN '82 Symp. on Compiler Construction, Boston, Massachusetts*, pages 261–275, June 1982. SIGPLAN Notices, Vol. 17, No. 6.

[Bon91] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17 (Selected papers of ESOP '90, the 3rd European Symposium on Programming)(1-3):3–34, Dec. 1991.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proc. ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation, Toronto, Ontario*, pages 278–292, ACM, ACM Press, June 1991.

[CU90] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. In *Proc. ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation (PLDI), White Plains, New York*, pages 150–164, ACM Press, June 1990.

[Dyb87] R. K. Dybvig. *The Scheme Programming Language.* Prentice-Hall, 1987.

[FSS75] S. Freudenberger, J. Schwartz, and M. Sharir. Experience with the SETL optimizer. *ACM TOPLAS*, 5(1):26–45, Jan. 1975.

[Gol91] B. Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proc. ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation (PLDI), Toronto, Ontario*, pages 165–176, ACM Press, June 1991.

[Gom90] C. Gomard. Partial type inference for untyped functional programs (extended abstract). In *Proc. LISP and Functional Programming (LFP), Nice, France*, July 1990.

[Hen91] F. Henglein. Efficient type inference for higher-order binding-time analysis. In *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA), Cambridge, Massachusetts*, pages 448–472, Springer, Aug. 1991. Lecture Notes in Computer Science, Vol. 523.

[Hen92] F. Henglein. Dynamic typing. In *Proc. European Symp. on Programming (ESOP), Rennes, France*, pages 233–253, Springer, Feb. 1992. Lecture Notes in Computer Science, Vol. 582.

[HL] R. Harper and M. Lillibridge. The standard ml of new jersey implementation of callcc is not type safe. Announcement on TYPES mailing list, July 8, 1991.

[JGZ88] R. Johnson, J. Graver, and L. Zurawski. TS: an optimizing compiler for Smalltalk. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Conf. Proc., San Diego, California*, pages 18–26, ACM Press, Sept. 1988.

[JM76] N. Jones and S. Muchnick. Binding time optimization in programming languages: some thoughts toward the design of the ideal language. In *Proc. 3rd ACM Symp. on Principles of Programming Languages*, pages 77–94, ACM, Jan. 1976.

[JM82] N. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 66–74, Jan. 1982.

[Jr92] J. Jø rgensen. Generating a compiler for a lazy language by partial evaluation. In *Proc. 19th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programmin gLanguages (POPL), Albuquerque, New Mexico*, pages 258–268, ACM Press, Jan. 1992.

[KKR*86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: an optimizing compiler for scheme. In *Proc. SIGPLAN '86 Symp. on Compiler Construction*, pages 219–233, 1986.

[KU80] M. Kaplan and J. Ullman. A scheme for the automatic inference of variable types. *J. ACM*, 27(1), Jan. 1980.

[LW91] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *Proc. 18th Annual ACM Symp. on Principles of Programming Languages, Orlando, Florida*, pages 291–302, ACM, ACM Press, Jan. 1991.

[MK90] K. Ma and R. Kessler. TICL – a type inference system for common lisp. *Software Practice & Experience*, 1990.

[MTH90] R. Milner, M. Tofte., and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Myc84] A. Mycroft. Dynamic types in statically typed languages. Aug. 1984. Unpublished manuscript, 2nd draft version.

[Pet89] J. Peterson. Untagged data in tagged environments: choosing optimal representations at compile time. In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 89–99, ACM Press, Sept. 1989.

[Ple91] U. Pleban. Compilation issues in the screme implementation for the 88000. In P. Lee, editor, *Topics in Advanced Language Implementation*, chapter 7, pages 157–188, MIT Press, 1991.

[Sch91] *IEEE Standard for the Scheme Programming Language.* IEEE Computer Society, IEEE std 1178-1990 edition, May 1991. Sponsored by the Microprocessor and Microcomputer Standards Subcommittee.

[Ses89] P. Sestoft. Replacing function parameters by global variables. In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 39–53, ACM Press, Sept. 1989.

[SH87] P. Steenkiste and J. Hennessy. Tags and type checking in LISP: hardware and software approaches. In *Proc. 2nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, Oct. 1987.

[Shi88] O. Shivers. Control flow analysis in Scheme. In *Proc. SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 164–174, ACM Press, June 1988.

[Shi91a] O. Shivers. Data-flow analysis and type recovery in scheme. In P. Lee, editor, *Topics in Advanced Language Implementation*, chapter 3, pages 47–88, MIT Press, 1991.

[Shi91b] O. Shivers. The semantics of scheme control-flow analysis. In *Proc. Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), New Haven, Connecticut*, pages 190–198, ACM Press, June 1991.

[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.

[Ste84] G. Steele. *Common LISP – The Language.* Digital Press, 1984.

[Ste91] P. Steenkiste. The implementation of tags and run-time type checking. In P. Lee, editor, *Topics in Advanced Language Implementation*, chapter 7, pages 157–188, MIT Press, 1991.

[Tar83] R. Tarjan. *Data Structures and Network Flow Algorithms.* Volume CMBS 44 of *Regional Conference Series in Applied Mathematics*, SIAM, 1983.

[Ten74] A. Tenenbaum. *Type Determination for Very High Level Languages.* Technical Report NSO-3, Courant Institute of Mathematical Sciences, New York University, 1974.

[Tha90] S. Thatte. Quasi-static typing. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 367–381, ACM, Jan. 1990.

[Tof90] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, Nov. 1990.

[VP89] S. Vegdahl and U. Pleban. The runtime environment for Screme, a Scheme implementation on the 88000. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS), Boston, Massachusetts*, pages 172–182, April 1989. SIGPLAN Notices, Vol. 24, Special Issue May 1989.

[Wal88] K. Walker. *A Type Inference System for Icon.* Technical Report 88-25, University of Arizona, July 1988.

[Wei86] G. Weiss. *Recursive Data Types in SETL: Automatic Determination, Data Language Description, and Efficient Implementation.* Technical Report 201, Courant Institute of Mathematical Sciences, New York University, March 1986.