Optimizing Floating Point Operations in Scheme

W.F. Wong

Department of Computer Science,

National University of Singapore,

Lower Kent Ridge Road,

Singapore 119260.

Republic of Singapore.

email: wongwf@comp.nus.edu.sg

Abstract

It is well known that dynamic typing in languages like Lisp is costly in terms of performance. Besides the cost of tag checking, the other major source of inefficiency comes from the need to place and retrieve data from dynamically allocated objects, i.e. *boxing* and *unboxing*. This makes it unacceptable in general to write numerical code in Lisp. Such programs involve "tight" loops in which boxing, unboxing and tag checking will dominate the computation time. With advances in the compilation of Lisp programs, it has been suggested that type checking and inference can be used to alleviate the problem. In this paper we shall examine a sub-problem, namely using type inference to aid compilation of numerical intensive Lisp code. A type inference algorithm for floating point operations will be described. This has been implemented in a Scheme compiler. Implementation issues and performance results on fairly large numerical code will also be reported. The results suggest that significant performance

gains can be obtained. It is our hope that as an augmentation to other general type inferencing scheme, it will contribute towards the realization of highly optimizing Scheme compilers.

Keywords: Scheme, tag optimization, type inference, compilers

1 Introduction.

Lisp is a dynamically typed language and it is well known that tag handling consumes a substantial amount of time in the execution of Lisp programs [1]. Generally, this can be attributed to two classes of operations:

- *Boxing and unboxing*: these are operations to, respectively, place and retrieve ("raw") data in allocated and tagged objects and return the appropriate references. Boxing is necessitated by polymorphism and the need for uniform representation. It can also be found in statically typed languages. In many Lisp implementations, bits in integers, for example, are sacrificed for tags. For example, one popular trick is to use the lowest two bits of a 32 bit word to indicate if the word is an integer or a pointer. While this does not involve additional memory allocation, for the purpose of this paper, we shall consider it to be a form of boxing as shift operations are needed to recover the actual integer.

- *Dynamic tag checking*: this refers to the need to ascertain the types of operands before invoking the appropriate operators.

This has prompted research into the use of type inference in the compilation of Lisp programs [2]. The basic strategy is based on the following approach:

1. extract as much type information from a Lisp program as possible either by some automated means or with user assistance in the form of type declarations;

2. perform type inference either within functions only (*local type inference*) or both within and between functions (*global type inference*);

3. for those data items whose types are completely determined, they may be unboxed and the operators changed appropriately.

In the past, research focused on the application of this technique to all data in a Lisp program. In this paper, we shall focus on a subproblem which turns out to be easy to solve and reaps significant performance gains in many situations. Specifically, the problem addressed is that of type inference over arithmetic operators. This paper describes an algorithmic approach for the optimization of floating point operations in Scheme programs. The approach is based on *control flow analysis.*[3]

```
(do ((i 0 (+ i 1)))
    ((>= i n))
  (vector-set! X i (+ (vector-ref Y i)
                      (* A (vector-ref X i)))))
```

Figure 1: Fragment of the SAXPY/DAXPY Loop in Scheme.

We shall now use a small example to hint at the potential gains involved. Fig. 1 shows a fragment of the SAXPY/DAXPY loop commonly found in many numerical application, here coded in Scheme. At least in theory, the vector accesses can be made almost as efficient as that of C or Fortran. The main problem is in the arithmetic operations. Take for example the multiplication operator. Because of dynamic typing, it is necessary to

check the types of A and X_i and then decide on the appropriate multiplication operations to use *in each iteration of the loop*. Furthermore, after the multiplication, it is necessary to *box* the result before the addition operation can begin. The addition operator will have to do the same operations of checking the types of Y_i and the result returned by the multiplication, decide on the appropriate operation to use and then box its result before returning. There are therefore four unbox and two box operations during *each* iteration of the loop. In all practical implementations, this overhead will cost significantly in terms of performance especially when compared to the two actual arithmetic operations. Since numerically intensive programs spend a lot of time in loops like this, the overall overhead of boxing and unboxing is very significant. If we can decide, during compile time, the actual arithmetic operators to be used and use unboxed data instead of boxed ones, the saving will therefore be significant.

In our work, we made the following assumptions:

A1: *Type inference is applied only as an optimization.* The algorithm was designed and implemented with the assumption that the input program is correct.

A2: *Numbers are either fixed point integers and (double precision) floating point numbers.* This reduces the overloading overheads of the arithmetic operators and simplifies the implementation. In particular, "bignums" are not considered. It is very difficult to detect overflow of integers into "bignums" during compile time and this severely limits the deployment of machine representations as unboxed values and thus the use of machine-supported arithmetic operations.

A3: *Minimal user intervention.*

In section 2, we shall outline the algorithm. This is followed by the discussion of how it was implemented in a public domain Scheme compiler. In section 3, performance results will be reported and discussed. This is followed by a brief survey of related work done to highlight the contribution of this paper and the conclusion.

2 Floating Point Type Inference.

The type inference requires two passes through the source program. The basic steps in the type inference algorithm is as follows:

1. Type information extraction and program annotation;
2. Type inference and propagation;
3. Program transformation.

The details of each step will now be described.

2.1 Type Information Extraction and Program Annotation.

The first step of the algorithm is an attempt to associate each expression, E, of the source program with a type variable, τ_E. The following are used in the description:

`Sexp`	=	Scheme expressions
`Aexp`	=	Annotated Scheme expressions
`TypeVar`	=	type variables
`Var`	=	program variables
`FuncLab`	=	labels for function definitions
`SimpleType`	=	$\{\top,$ `Number`, `Integer`, `Float`, `Other`, $\bot\}$
`ListOpType`	=	(`op` $\langle$ `TypeVar`$^+$ $\rangle$) where `op` is a list operator
`FuncType`	=	(`func`$_{\texttt{FuncLab}}$ $\langle$ `TypeVar`* $\rangle$ $\longrightarrow$ `TypeVar`)
`AppType`	=	(`apply FuncLab` $\langle$ `TypeVar`* $\rangle$)
`CallCCType`	=	(`call/cc FuncLab`)
`TypeSeq`	=	`SimpleType` | `TypeVar` | `ListOpType` | `FuncType` | `AppType` | `CallCCType`
`TypeBind`	=	`TypeVar` $\mapsto$ `TypeSeq`$^+$
`VarTypeBind`	=	`Var` $\mapsto$ `TypeVar`$^+$

The simple types are $\{\top$, `Number`, `Integer`, `Float`[1], `Other`, $\bot\}$. The simple type '$\bot$' is used to represent the 'unknown type' while '$\top$' represents 'a mixture of other simple types.' In the above, '*' denotes "zero or more occurrences in a set", '$^+$' denotes "one or more occurrences in a set", while '$\langle\cdots\rangle$' indicates an ordered set, i.e. a sequence. Fig. 2 shows the type lattice we used. Since we are primarily interested in the domain of numbers, other Scheme types such as characters, strings etc. are grouped together under the type `Other`. However, we believe that our approach can be extended to these other types to get more refined results.

[1]The entire paper is based on the assumption that numbers are either fixed point integers or floating point numbers. 'Bignum's are assumed not available.

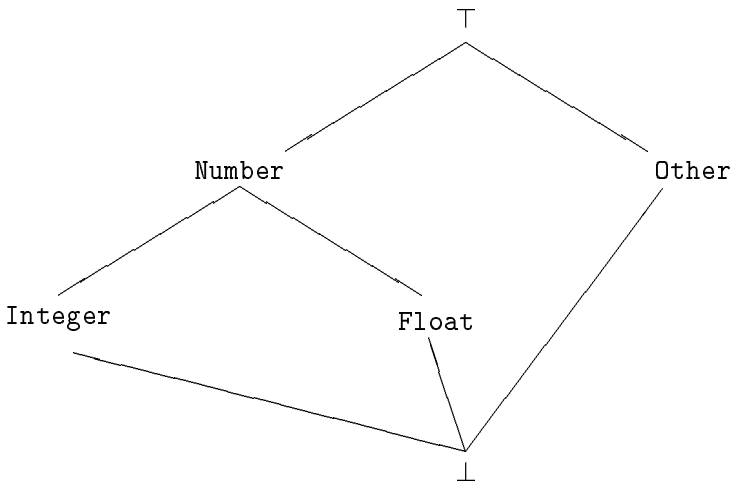

Figure 2: Lattice of Simple Types.

The annotation function, $\widehat{\mathcal{A}}$, of the following type,

$$\widehat{\mathcal{A}} \;:\; (\texttt{Sexp} \times \texttt{TypeBind}^* \times \texttt{VarTypeBind}^* \times \texttt{VarTypeBind}^*) \longrightarrow$$
$$(\texttt{Aexp} \times \texttt{TypeBind}^* \times \texttt{VarTypeBind}^* \times \texttt{VarTypeBind}^*)$$

accepts a Scheme expression of type `Sexp`, the set of type variable bindings of type `TypeBind`, the local environment of type `VarTypeBind` and the global environment also of type `VarTypeBind`. The function returns a four-tuple, consisting of an annotated Scheme expression of type `Aexp`, the modified set of type variable bindings, local environment and global environment.

Each annotated Scheme expression begins with a type variable. The basic idea of annotation is to link this type variable with either a simple type, composite type or other type variables. This is done using the semantics of Scheme such that after type inference, it will correctly hold the type of the result of the expression's evaluation. For some simple operators, this type variable can be bound to a simple type. This is our equivalent of the

type extraction idea first given by Shivers[4]. To do this, it is necessary to have program variables consistently bound to type variables. For instance, if the program variable x is used in two places and if they refer to the same variable (according to the semantics of Scheme), then we want to reflect this in the annotation. However, it is possible that the same variable at different places in the program is assigned values of different types. A set of result type variables is then maintained so that this may be settled during type inferencing. This is essentially a form of *monovariant* flow analysis [5] where each variable and subexpression is associated with a set of abstract values. In contrast, *polyvariant* analysis [5] uses *program contours* to distinguish multiple abstract values associated with a single type variable.

Fig. 3 shows the first part of $\widehat{\mathcal{A}}$ which annotates constants and variables. In the description '$\mathcal{E}[x \mapsto \tau]$' is the functional update or extension of an environment $\mathcal{E}$. In Fig. 3, $\mathcal{T}$ is the type binding environment which hold the bindings of type `TypeSeq` to each type variable. The local environment, $\mathcal{E}$, holds the set of type variables associated with each program variable lexically visible at a particular program point. The global environment, $\mathcal{G}$ is similar to $\mathcal{E}$ except that it holds the bindings for global program variables. To annotate a variable, the local environment, $\mathcal{E}$, will be checked to see if there is already a type variable bound to it. If not, the global environment, $\mathcal{G}$, will then be searched. If it is also not found there, a new binding associating the variable with $\perp$ will be made in the global environment.

Fig. 4 shows how `set!` expressions are annotated. It proceeds in a manner similar to that for variables except that instead of $\perp$, the variable is associated with the type variable of the expression e.

Vectors and their operators are annotated according to Fig. 5. Essentially, the type of a vector will be dependent on each of its elements. This is handled by associating the type of a vector to a set of type variables. Furthermore, each `vector-set!` to a vector will contribute a new type variable, namely that of the source expression, to the set. At type inference, the type of a vector will be determined by taking the *least upper bound* of the types represented by each of the type variables according to the type lattice.

2.1.1 Operators.

The annotation of operators is shown in Fig. 6. Type extraction for operators producing integers and floating point numbers is shown in the figure. For list operations, the actual operator used is kept as part of the type information. During type inference, the operation is attempted on the type sequence.

2.1.2 Conditionals.

Fig. 7 shows the annotation for conditionals. The type sequences of both arms of a conditional are maintained as a set, namely $\{\tau_t \ \ \tau_f\}$. As an enhancement, an auxiliary function $\widehat{\mathcal{A}}'$ checks the conditional test and if it is either `integer?` or `real?`, then the type variable concerned is forced to the respective type so that more information is available during the annotation of the body.

2.1.3 Sequence, let and lambda.

As shown in Fig. 8, the type variable of a sequence is bound to the type variable of the last expression in the sequence, i.e. its type is exactly that of the last expression. For `let`, the local environment must be extended by the newly introduced local variables before the body can be annotated.

Functions are typed as $(\texttt{func}_l\ (\tau_0 \cdots \tau_k \longrightarrow \tau_l))$ where l is a unique label that identifies the definition of the function, $\tau_0, \cdots, \tau_k$ are the type variables associated with the formal parameters and τ_l is the type variable associated with the result. The body of the function is annotated as a sequence with the proper extension of the local environment by the local variables.

2.1.4 Function Applications.

Applications of explicit `lambda`s are straightforward (see Fig. 9) - each formal parameter is given the type of the corresponding actual argument before the body of the `lambda` is processed. The type of the application is the type of the last statement in the `lambda` body.

Non-`lambda` applications are typed as (possibly sets of) $(\texttt{apply}\ l\ \Psi)$ where l is the label of the function's definition and Ψ is a sequence of type variables corresponding to each of the actual arguments. The operator is first annotated. Should this be a variable that is undefined in both the global or local environment, a fresh function label is generated and the global environment is extended by assuming that the function take the same number of parameters as there are actual arguments with an unknown return type. If, however, the operator is not a variable, i.e. it is another expression, then an auxiliary function $\widehat{\mathcal{J}}$ is used to locate a set of function labels for the functions that the expression *may* evaluate to. The type of the application is then a set of application types indicating the set of possible types the result of the application may take. This method of handling function calls is similar to Shivers' 0CFA [4].

2.1.5 Continuations.

Calls with current continuation are typed as (possibly a set of) (`call/cc` l) where l is the label of the single parameter function which is the sole argument to a `call/cc`. In the case of an explicit `lambda` definition, a new function label, l say, is generated and used in the function type sequence. The type of the result of the `call/cc` expression is set to (`call/cc` l). The idea is that the type of the `call/cc` expression is determined by the type of the formal argument in the applications of l. In the case of a non-`lambda` argument to `call/cc`, $\widehat{\mathcal{J}}$ is invoked to find the set of possible functions that may be the argument during runtime. The type of the result of the `call/cc` expression is then set to a set of (`call/cc` i) where i is a member of the set returned by $\widehat{\mathcal{J}}$. The details are shown in Fig. 11.

2.2 Type Inference.

After the program has been annotated by calling $\widehat{\mathcal{A}}\,\langle \mathcal{P}, \emptyset, \emptyset, \emptyset \rangle$ where $\mathcal{P}$ is the sequence of Scheme expression that makes up the program, type inferencing may begin. Here type inferencing is equivalent to the computation of the type variables. Unlike program annotation, however, this is not quite a distinct phase. What is now possible is the computation for the value of any type variable. However, this value may still change, as will be apparent later, and the same inference algorithm may have to be invoked again to find its new value.

Fig. 12 shows the basic algorithm for computing the final value, in terms of simple types, of any type variable in a program. Note that when there is a set of possible final values for a type variable, we will make use of the type hierarchy given in Fig. 2 to obtain the final value which the type variable will contain. Note also how type variables which

result from list operations are obtained. The list operation is noted down during the annotation phase and is actually evaluated on the available arguments. Should this not be possible, the simple type $\perp$ is returned. It may be necessary sometimes to deal only with simple types. In this case, an operator $\widehat{\mathcal{K}}$ can be defined as follows:

$\widehat{\mathcal{K}}$: `SimpleType`* $\longrightarrow$ `SimpleType`

$$\widehat{\mathcal{K}}\,\langle t\rangle = \begin{cases} t & \text{if } t \text{ is a } \texttt{SimpleType} \\ \perp & \text{otherwise} \end{cases}$$

will force lists to the simple type `Other`.

The existence of circularity in control flow analysis type inference algorithms has been noted and studied by Nielson and Nielson [3]. This happens, for example, when in an attempt to infer the type of type variable τ, we run into a recursive occurrence of itself in one of the type sequences or $\tau \mapsto \Gamma$ is in $\mathcal{T}$ and $\tau \in \Gamma$. This implies recursion at the source level. Termination is ensured in our inference algorithm by keeping track of all the type variables seen in the inference process so far by the set $\mathcal{M}$ of $\widehat{\mathcal{I}}$. When an attempt is made to infer a type variable which has already been encountered, $\perp$ is returned as the result of the inference on the *last* occurrence of the type variable and the inference process continues from that point on. This is done instead of terminating the entire inference and returning $\top$ as the overall result because there is a chance that further inference using this new binding *may* yield more precise results.

An important feature in our scheme is the automatic specialization of functions based on the types of the arguments of the calls. A similar idea was pursued in the compiler for Smalltalk-like language SELF [6]. The algorithm that performs the specialization is given in Fig. 13. The basic idea is to collect all the calls to a (named) function and then partition the calls according to the data types of the arguments. Based on the partitions,

new instances of the function, appropriately renamed, will be created. The names and data types of the arguments are noted so that during the transformation phase, changes can be made to the function calls. The idea behind doing this is that once the data types of the arguments can be fully determined, it will create opportunites to optimize operations in the function body. We have the following:

Theorem 1 *Let $\mathcal{T}$ be the type binding after the function specialization phase, then for all type variables $\tau \in \mathcal{T}$, $\widehat{\mathcal{K}}\left\langle \widehat{\mathcal{I}}\left\langle \tau, \mathcal{T}, \emptyset \right\rangle \right\rangle \in$* `SimpleType`.

Proof: The proof is quite straightforward. First, we note that the type bindings $\mathcal{T}$ produced by the annotation algorithm are strictly of the form `TypeBind` for any valid Scheme program. This is possible because of the 'catch-all' that produces $\tau \mapsto \perp$ whenever there are unknown operators. For each type variable, the inference algorithm will then produce either a `SimpleType` or a `TypeSeq` which is then reduced by $\widehat{\mathcal{K}}$. □

For correctness, we informally claim that given a type variable, the inference algorithm will at worst report an upper bound, with respect to the type hierarchy of Fig. 2, of its 'true' type.

2.3 Program transformation.

Given that we can infer type information, we are now in a position to optimize the program by means of source-level code transformation. This is outlined in Fig. 14.

For the most part, program transformation performs the inverse operation of program annotation, i.e. it strips the type variables from the annotated expressions resulting in an executable Scheme program. The two major differences are in the handling of arithemtic expressions and function calls. For the latter, it may be necessary to rename the call based

on the data types of the arguments inferred.

Using the type information available, it is possible to specialize the arithmetic operations based on the inferred data types of the arguments. In Fig. 14, we assume the availability of 3 operators:

- `int-op` : the integer arithmetic operators, i.e. `int-+`, `int--` and `int-*`, which assume that the inputs are integers and will not do tag checking on their inputs;
- `fp-op` : the floating point arithmetic operators, i.e. `fp-+`, `fp--`, `fp-*` and `fp-/`, which assume that the inputs are floating point numbers and will not do tag checking on their inputs;
- `exact->inexact` : converts a number, which may be an integer or a floating point number, to a floating point number;

In other words, the specialized arithmetic expression will do tag checking and conversion explicitly only in cases where the data type of the argument is uncertain.

The program transformation may introduce new type bindings or change existing ones. This will continously modify the set of type bindings on which the type inference algorithm will work on. It is fairly easy to see that the termination of the inference algorithm is not affected. An inductive argument can be used to argue for the correctness of the transformation. Assuming that the inference algorithm is correct for a given annotated program and its associated type binding, then by the principle that, for arithmetic operators considered, the result of its application would be a floating point number should any of its operand be a floating point number.

3 Implementation Issues.

We have implemented the above algorithms in Scheme→C [7] and have proceeded to modify the Scheme→C compiler to support the new arithmetic operators. It turns out that efficient implementations of these operators are crucial for performance. In addition, some other optimizations are done.

In Scheme→C , the arithmetic operators are implemented in two levels. First, as a macro, the compiler will issue code which will check if both arguments are integers. If they are, the operation can be performed immediately. If not, a call will be made to the corresponding routine in the runtime library. Fig. 15 shows how the '+' operator is implemented in the Scheme→C runtime library. By checking the tag of the two arguments, the appropriate add operation together with the necessary type conversion (here represented as macros) will be performed.

In our implementation, we augment the original Scheme→C with new arithmetic operations that will not do any tag checking. This is similar to Soft Typing [8] but with explicit coercion where necessary. Only in the face of uncertainty will the type inference program emit explicit type checks.

Another important optimization has to do with intermediate results and can be illustrated by the following example:

```
(* (+ A B) (- C D))
```

This would translate to (1) unbox A, (2) unbox B, (3) add, (4) box the result, (5) unbox C, (6) unbox D, (7) subtract, (8) box the result, (9) unbox the result of `(+ A B)`, (10) unbox the result of `(- C D)`, (11) multiply, and (12) box the result. Immediately we see that operations (4), (8), (9) and (10) can be eliminated if the results of the addition and

the subtraction had stayed unboxed. Not only is the number of tag checking and boxing operations reduced, so too are the necessary memory accesses. Accordingly, the type inference program will emit new operations for intermediate computation that in addition to working with unboxed data will produce unboxed results in special variables which can then be immediately used by the operator at the next level.

4 Performance Results.

Having given the type inference algorithm and outlining its implementation in Scheme→C we have evaluated the performance gains using three numerically intensive programs. They are:

- **Linpack**. The famous Gaussian elimination with partial pivoting benchmark was translated from its C version into Scheme. The double precision version was used as Scheme→C uses double precision internally.

- **mp3d**. This is the molecular collision simulation program that is part of the SPLASH [9] suite of benchmarks. The whole code was translated from C into Scheme. The type inferencing was applied only to the code for advancing the simulation which accounts for the bulk of the computation.

- **Simplex**. This is a program that solves a system of inequalities using the simplex method.

The performance results are given in Table 1. The tests were conducted on a DEC 3000/300L workstation with a 100 MHz A21064 Alpha CPU running OSF/1 with only one user logged in.

In Table 1, the column for **Opt-1** indicates the performance with type inferencing only whereas the column for **Opt-2** indicates the combined type inferencing together with the elimination of the boxing and unboxing of intermediate results as mentioned in Section 3. The 'improvement' is defined by (|Speed of original − Speed of optimized version|) ÷ Speed of original.

The results indicate that a performance gain of 5% to 25% can be obtained depending on the application. With **Opt-2**, there is an additional gain. Eliminating unnecessary boxing and unboxing operations also reduces the overall memory requirements. This is shown by the reduction in the number of calls to the garbage collector (except for one case in **mp3d** which we are unable to explain). This allowed **Linpack** to be executed with n up to 400. This partially accounts for the performance improvement. In the best case, we obtained more than twice the original performance.

4.1 A Note on Function Calls.

The careful reader would have observed that the number of calls to the garbage collector is different even between the original and the **Opt-1** version which introduces floating point operations explicitly. Upon closer scrutiny, this revealed an important characteristic of our algorithm. Recall that the algorithm will examine each function call and based on the type information inferred will instantiate new versions of the same function. For example, let `A(x)` be a function which takes one argument. Suppose there are two calls to `A`, one with `A(s)` and one with `A(t)`. In general, we cannot guarantee that the type of `s` will be identical to that of `t`. If we find that `s` is a floating point number, then we can optimize the body of `A` accordingly. However, it may turn out that we are unable to say

[†]Crashed after 283 calls to the garbage collector.

the same about `t`. In this case, we need two versions of `A` to cater for both calls.

Unfortunately, the above can have a potentially negative impact on *tail recursive calls.* Scheme→C optimizes tail recursive calls by replacing them with `goto` statements. The above phenomenon can prevent such an optimization from taking place. For example, both `A(s)` and `A(t)` may be tail recursive calls. However, after specialization, at best one of them will remain tail recursive; the other being instantiated to call a different version of `A`. The result is a potential loss in performance and an increase in memory requirements (which is reduced by tail recursion optimizations) and therefore garbage collection activities. This explains the poor performance of the `Simplex` code in which exactly such a situation occurred. It should be possible to detect such a situation and then disable the specialization although this is not done currently.

5 Related Works.

The problem of tag optimization has also received constant attention from Lisp and functional programming language researchers. Tag optimization in functional programming language is done in the context of polymorphic functional programming languages [10]. Most Lisp compilers such as ORBIT [11] or Screme [12] will do local tagging optimizations [1]. TICL [13] is a type system for Common Lisp but it relies on user declarations.

Type inference in high level dynamic typed languages is not a new problem. Early works on this problem include [19], [14] and [15]. Their approaches are mainly based on data flow analysis. Shivers' [4] and Henglein's [2] works are the inspiration for ours. In particular, our approach is similar to Shivers' 0CFA [4], which does control flow analysis on distinct call sites, except that

- we do not work with a continuation-passing flavor of Scheme;

- we do not keep explicit call-sites;
- we use the type information for specialization as well as tag optimizations;

Henglein tackled the problem of tag optimization from a global perspective without resorting to control flow analysis. His method was particularly effective for eliminating tag checking in list operations. In EuLisp [16], all these works were combined into a practical system.

Berlin [17] uses partial evaluation and symbolic manipulation to specialize numerical programs. His method requires the symbolic execution of the input program at compile time whereas ours only does this for list operations. He also did not consider the problem of boxing and unboxing, although he did briefly mention the potentials for performance improvement. This was taken up by SUA [18] which optimizes boxing. However, uniform representation is sacrificed. Furthermore, continuations and function specialization are not considered.

Our work differs from these earlier works in that

- the focus is on the arithmetic operators;
- the type inference is done from a global perspective;
- we investigated the necessary modifications that must be done internally to the compiler;
- we have implemented and tested the algorithm on medium size applications;

Our work should be seem as an augmentation for the suite of type optimization techniques that have been proposed.

6 Conclusion.

In this paper we described a simple type inferencing mechanism applicable to floating point computation in full Scheme - with side-effect operators, vectors and continuations. Through a three step process of annotation, inference and transformation, floating point operations are optimized. This improves the performance of floating point Scheme code by reducing the type checking operations, especially in numerical loops. Memory requirements are also reduced thereby reducing the number of times of garbage collector have to be invoked.

We have implemented the algorithm and tested it out by modifying the Scheme→C compiler. We have found that on some applications we were able to double the performance. As a reference, the optimized C version of the double precision Linpack benchmark did a little less than 10 Mflops. The Scheme version optimized by our type inference system achieved a little better than 1/20 of this performance. Our aim is to reduce the performance gap between Lisp and other procedural languages in every aspect of general purpose computing. Much remains to be done but we hope that the algorithm reported in this paper will be a contribution towards this goal.

Acknowledgement

I would like to thank Siau-Cheng Khoo for his detail reading and critique of this paper.

References

[1] P. A. Steenkiste, 'The Implementation of Tags and Run-time Type Checking', *Topics in Advanced Language Implementation*, P. Lee ed., pp. 3-24. MIT Press 1991.

[2] F. Henglein, 'Global Tagging Optimization by Type Inference', *Proc. of ACM Symp. on Lisp and Functional Programming*, pp. 205-215. 1992.

[3] F. Nielson and H.R. Nielson, 'Infinitary control flow analysis: a collecting semantics for closure analysis', *Proc. of ACM Symp. on Principles of Prog. Lang.*, pp. 332-345. 1997.

[4] O. Shivers, 'Data-flow analysis and type recovery in Scheme.' *Topics in Advanced Language Implementation*, P. Lee ed., pp. 47-88. MIT Press 1991.

[5] S. Jagannathan and A. Wright, 'Flow-directed Inlining', *Proc. ACM Symp. on Prog. Lang. Design and Impl.*, pp. 193-205. 1996.

[6] C. Chambers and D. Ungar, 'Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object Oriented Programming Language,' *Proc. ACM Symp. on Prog. Lang. Design and Impl.*, pp. 146-160. 1989.

[7] J. F. Bartlett, 'SCHEME->C : a portable Scheme-to-C Compiler', *DEC Western Research Lab. Research Report 89/1*, Jan 1989.

[8] A. K. Wright and R. Cartwright, 'A Practical Soft Type System for Scheme', *ACM Trans. on Prog. Lang. & Sys.*, vol. 19, no. 1, pp. 87-152. 1997.

[9] J. P. Singh, W.-D. Weber and A. Gupta, 'SPLASH: Stanford Parallel Applications for Shared-Memory', *Computer Architecture News*, vol. 20, no. 1, pp. 5-44. 1992.

[10] X. Leroy and P. Weis, 'Polymorphic type inference and assignment', *Proc. 18th ACM Symp. on Principles of Prog. Lang.*, pp. 291-302. Jan 1991.

[11] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin and N. Adams, 'ORBIT: An Optimizing Compiler for Scheme', *Proc. of SIGPLAN'86 Symp. on Compiler Construction*, pp. 219-233. 1986.

[12] U. F. Pleban, 'Compilation issues in the Screme implementation for the 88000.' *Topics in Advanced Language Implementation*, P. Lee ed., pp. 157-188. MIT Press 1991.

[13] K. Ma and R. Kessler, 'TICL - a type inference system for Common Lisp', *Software Practice and Experience*, vol. 20, no. 6, pp. 593-623. Jun 1990.

[14] N. Jones and S. Muchnick, 'Binding time optimization in programming languages: Some thoughts toward the design of the ideal language.' *Proc. of 3rd ACM Symp. on Principles of Prog. Lang.*, pp. 77-94. Jan 1976.

[15] M. Kaplan and J. Ullman, 'A scheme for the automatic inference of variable types', *Journal of ACM*, vol. 27 no. 1, pp. 128-145. Jan 1980.

[16] A. Kind and H. Friedrich, 'A Practical Approach to Type Inference for EuLisp', *Lisp and Symbolic Comp.*, vol. 6, pp. 159-176. 1993.

[17] A. A. Berlin, 'A Compilation Strategy for Numerical Programs based on Partial Evaluation', *MIT AI Lab. AI-TR 1144*, Jul 89.

[18] M. Serrano and M. Feeley, 'Storage Use Analysis and its Applications', *Proc. of ACM International Conf. on Functional Programming*, pp. 50-61. 1996.

[19] A. Tenebaum, 'Type Determination for very high level languages', *Courant Inst. of Math. Sc. TR NSO-3*. 1974.

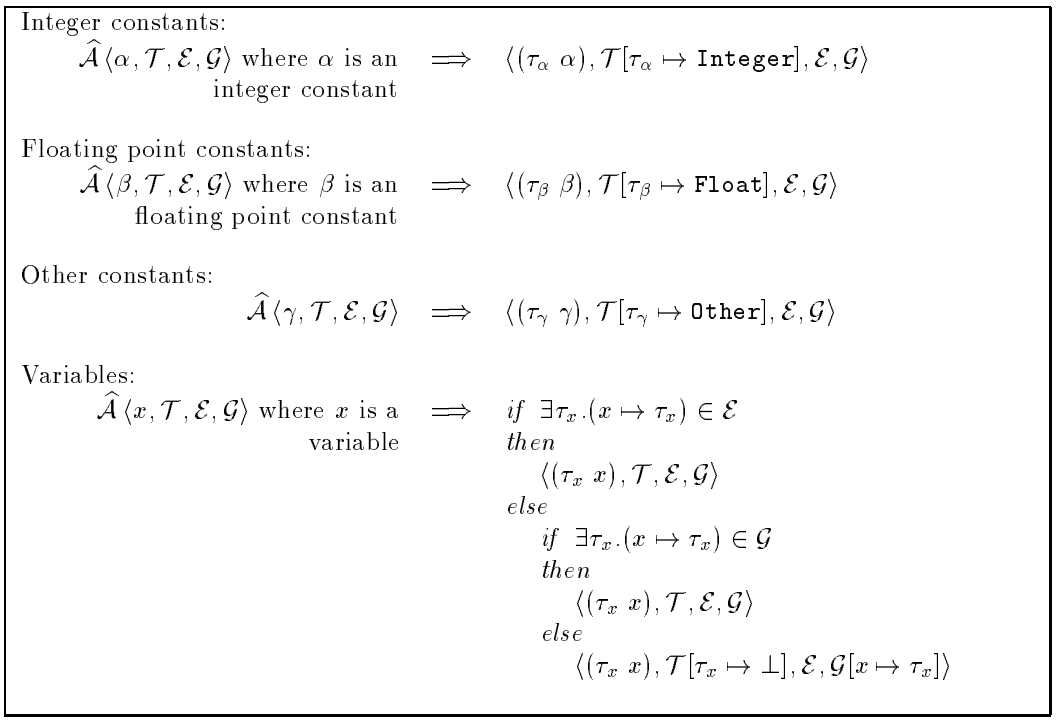

Figure 3: Annotation Algorithm for constants and variables.

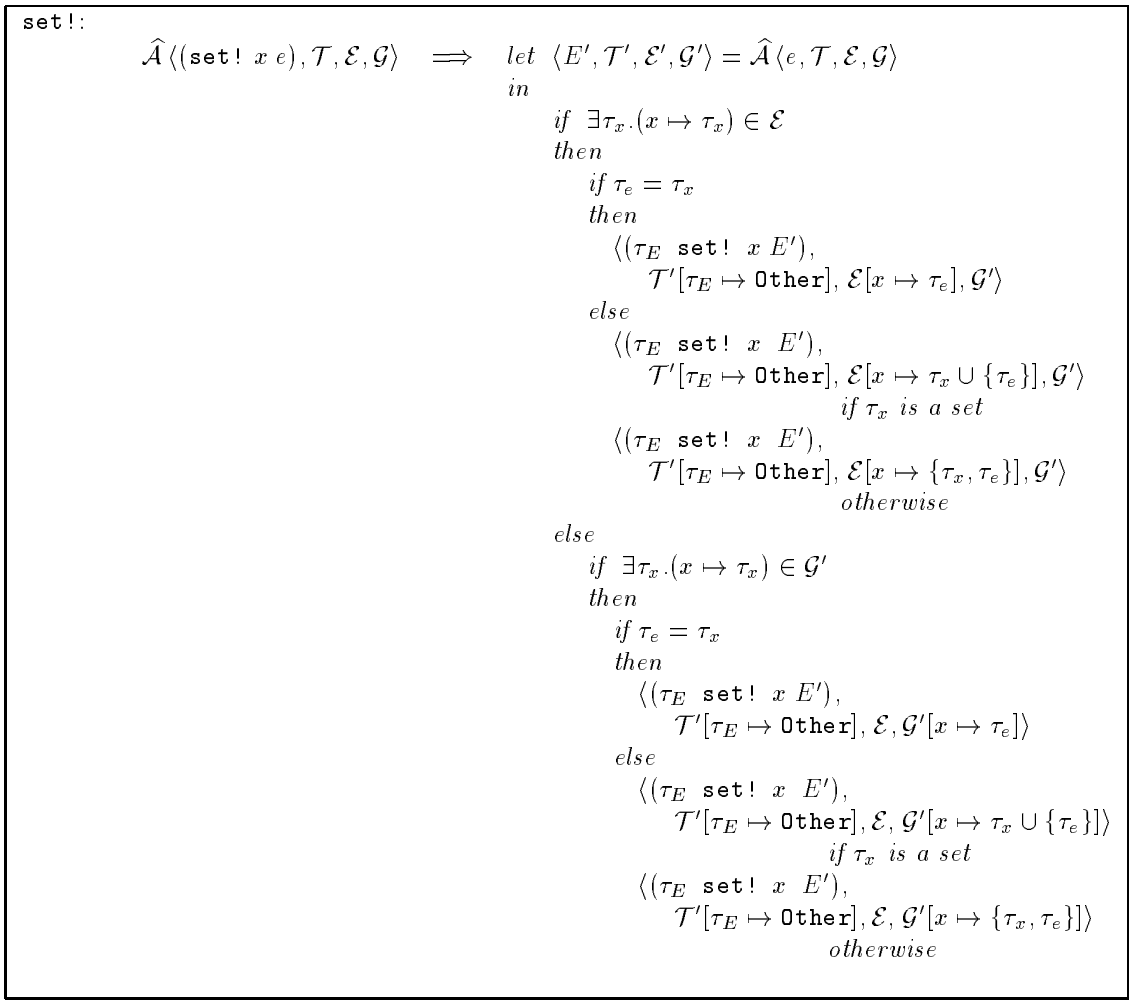

Figure 4: Annotation Algorithm for `set!`.

Vector constants:

$$\widehat{\mathcal{A}}\langle \texttt{\#}(\{e_i\}^*), \mathcal{T}, \mathcal{E}, \mathcal{G}\rangle \implies \begin{array}{l} \mathit{let}\ \left\{\langle \mathcal{S}_i,\ \mathcal{T}_i,\ \mathcal{E}_i,\ \mathcal{G}_i\rangle\ =\ \widehat{\mathcal{A}}\langle e_i,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G}\rangle\right\}^* \\ \quad \mathcal{T}' = \mathcal{T} \cup (\bigcup_{i=0} \mathcal{T}_i) \\ \quad \mathcal{G}' = \mathcal{G} \cup (\bigcup_{i=0} \mathcal{G}_i) \\ \mathit{in} \\ \quad \langle \texttt{\#}\,(\tau_E\ \ \{\mathcal{S}_i\}^*)\,, \\ \qquad \mathcal{T}'[\tau_E \mapsto \{\bot\} \cup (\bigcup_{i=0}\{\tau_{e_i}\})],\ \mathcal{E}, \mathcal{G}'\rangle \\ \text{where } \tau_{e_i} \text{ is the type variable} \\ \text{associated with } e_i \text{ in } \mathcal{S}_i \end{array}$$

Vector operations:

$$\widehat{\mathcal{A}}\langle(\texttt{vector}\ \{e_i\}^*), \mathcal{T}, \mathcal{E}, \mathcal{G}\rangle \implies \begin{array}{l} \mathit{let}\ \left\{\langle \mathcal{S}_i,\ \mathcal{T}_i,\ \mathcal{E}_i,\ \mathcal{G}_i\rangle\ =\ \widehat{\mathcal{A}}\langle e_i,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G}\rangle\right\}^* \\ \quad \mathcal{T}' = \mathcal{T} \cup (\bigcup_{i=0} \mathcal{T}_i) \\ \quad \mathcal{G}' = \mathcal{G} \cup (\bigcup_{i=0} \mathcal{G}_i) \\ \mathit{in} \\ \quad \langle(\tau_E\ \texttt{vector}\ \{\mathcal{S}_i\}^*)\,, \\ \qquad \mathcal{T}'[\tau_E \mapsto \{\bot\} \cup (\bigcup_{i=0}\{\tau_{e_i}\})],\ \mathcal{E}, \mathcal{G}'\rangle \end{array}$$

$$\widehat{\mathcal{A}}\langle(\texttt{make-vector}\ k), \mathcal{T}, \mathcal{E}, \mathcal{G}\rangle \implies \langle(\tau_E\ \texttt{make-vector}\ k), \mathcal{T}[\tau_E \mapsto \bot], \mathcal{E}, \mathcal{G}\rangle$$

$$\widehat{\mathcal{A}}\langle(\texttt{make-vector}\ k\ e_0), \mathcal{T}, \mathcal{E}, \mathcal{G}\rangle \implies \begin{array}{l} \mathit{let}\ \langle \mathcal{S}_0,\ \mathcal{T}_0,\ \mathcal{E}_0,\ \mathcal{G}_0\rangle\ =\ \widehat{\mathcal{A}}\langle e_0,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G}\rangle \\ \mathit{in} \\ \quad \langle(\tau_E\ \texttt{make-vector}\ k\ \mathcal{S}_0)\ ,\ \mathcal{T}_0[\tau_E \mapsto \tau_{e_0}],\ \mathcal{E}, \mathcal{G}_0\rangle \end{array}$$

$$\widehat{\mathcal{A}}\langle(\texttt{vector-ref}\ e_0\ k), \mathcal{T}, \mathcal{E}, \mathcal{G}\rangle \implies \begin{array}{l} \mathit{let}\ \langle \mathcal{S}_0,\ \mathcal{T}_0,\ \mathcal{E}_0,\ \mathcal{G}_0\rangle\ =\ \widehat{\mathcal{A}}\langle e_0,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G}\rangle \\ \mathit{in} \\ \quad \langle(\tau_E\ \texttt{vector-ref}\ \mathcal{S}_0\ k)\ ,\ \mathcal{T}_0[\tau_E \mapsto \tau_{e_0}],\ \mathcal{E}, \mathcal{G}_0\rangle \end{array}$$

$$\widehat{\mathcal{A}}\langle(\texttt{vector-set!}\ \ e_0\ k\ e_1), \mathcal{T}, \mathcal{E}, \mathcal{G}\rangle \implies \begin{array}{l} \mathit{let}\ \langle \mathcal{S}_0,\ \mathcal{T}_0,\ \mathcal{E}_0,\ \mathcal{G}_0\rangle\ =\ \widehat{\mathcal{A}}\langle e_0,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G}\rangle \\ \quad \langle \mathcal{S}_1,\ \mathcal{T}_1,\ \mathcal{E}_1,\ \mathcal{G}_1\rangle\ =\ \widehat{\mathcal{A}}\langle e_1,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G}\rangle \\ \quad \mathcal{T} = T_0 \cup T_1 \\ \quad \mathcal{G} = G_0 \cup G_1 \\ \mathit{in} \\ \quad \langle(\tau_E\ \texttt{vector-set!}\ \mathcal{S}_0\ k\ \mathcal{S}_1)\ ,\ \mathcal{T}[\tau_E \mapsto \{\tau_{e_0}, \tau_{e_1}\}],\ \mathcal{E}, \mathcal{G}\rangle \end{array}$$

Figure 5: Annotation algorithm for vector operators.

Arithmetic operators:

$$\widehat{\mathcal{A}}\langle(\texttt{op}\ e_0\ e_1), \mathcal{T}, \mathcal{E}, \mathcal{G}\rangle \implies \begin{array}{l} let\ \langle \mathcal{S}_0,\ \mathcal{T}_0,\ \mathcal{E}_0,\ \mathcal{G}_0\rangle = \widehat{\mathcal{A}}\langle e_0,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G}\rangle \\ \quad \langle \mathcal{S}_1,\ \mathcal{T}_1,\ \mathcal{E}_1,\ \mathcal{G}_1\rangle = \widehat{\mathcal{A}}\langle e_1,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G}\rangle \\ \quad \mathcal{T}' = \mathcal{T}_0 \cup \mathcal{T}_1 \\ \quad \mathcal{G}' = \mathcal{G}_0 \cup \mathcal{G}_1 \\ in \\ \quad \langle(\tau_E\ \texttt{op}\ \mathcal{S}_0\ \mathcal{S}_1)\ ,\ \mathcal{T}'[\tau_E \mapsto \texttt{Number}],\ \mathcal{E}, \mathcal{G}'\rangle \end{array}$$

where $\texttt{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$

Operations producing integers:

$$\widehat{\mathcal{A}}\langle(\texttt{op}\ e_0\ \{e_i\}^*), \mathcal{T}, \mathcal{E}, \mathcal{G}\rangle \implies \begin{array}{l} let\ \langle \mathcal{S}_0,\ \mathcal{T}_0,\ \mathcal{E}_0,\ \mathcal{G}_0\rangle = \widehat{\mathcal{A}}\langle e_0,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G}\rangle \\ \quad \left\{\langle \mathcal{S}_i,\ \mathcal{T}_i,\ \mathcal{E}_i,\ \mathcal{G}_i\rangle = \widehat{\mathcal{A}}\langle e_i,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G}\rangle\right\}^* \\ \quad \mathcal{T}' = \bigcup_{i=0} \mathcal{T}_i \\ \quad \mathcal{G}' = \bigcup_{i=0} \mathcal{G}_i \\ in \\ \quad \langle(\tau_E\ \texttt{op}\ \mathcal{S}_0\ \{\mathcal{S}_i\}^*)\ ,\ \mathcal{T}'[\tau_E \mapsto \texttt{Integer}],\ \mathcal{E}, \mathcal{G}'\rangle \end{array}$$

where $\texttt{op} \in \{$`char->integer`, `inexact->exact`, `length`, `modulo`, `quotient`, `string-length`, `vector-length`$\}$

Operations producing float pointing numbers:
Similar to the above except $[\tau_E \mapsto \texttt{Float}]$ for $\texttt{op} \in \{$`exact->inexact, sqrt`$\}$.

Operations producing other types:
Similar to the above except $[\tau_E \mapsto \texttt{Other}]$ for other operators.

List operations:

$$\widehat{\mathcal{A}}\langle((\texttt{op}\ e_0\ \{e_i\}^*), \mathcal{T}, \mathcal{E}, \mathcal{G}\rangle \implies \begin{array}{l} let\ \langle \mathcal{S}_0,\ \mathcal{T}_0,\ \mathcal{E}_0,\ \mathcal{G}_0\rangle = \widehat{\mathcal{A}}\langle e_0,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G}\rangle \\ \quad \left\{\langle \mathcal{S}_i,\ \mathcal{T}_i,\ \mathcal{E}_i,\ \mathcal{G}_i\rangle = \widehat{\mathcal{A}}\langle e_i,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G}\rangle\right\}^* \\ \quad \mathcal{T}' = \bigcup_{i=0} \mathcal{T}_i \\ \quad \mathcal{G}' = \bigcup_{i=0} \mathcal{G}_i \\ in \\ \quad \langle(\tau_E\ \texttt{op}\ \mathcal{S}_0\ \{\mathcal{S}_i\}^*)\ ,\ \mathcal{T}'[\tau_E \mapsto (\texttt{op}\ \tau_{e_0}\ \{\tau_{e_i}\}^*)],\ \mathcal{E}, \mathcal{G}'\rangle \end{array}$$

where `op` is a list operation.

Figure 6: Annotation algorithm for arithmetic and list operators.

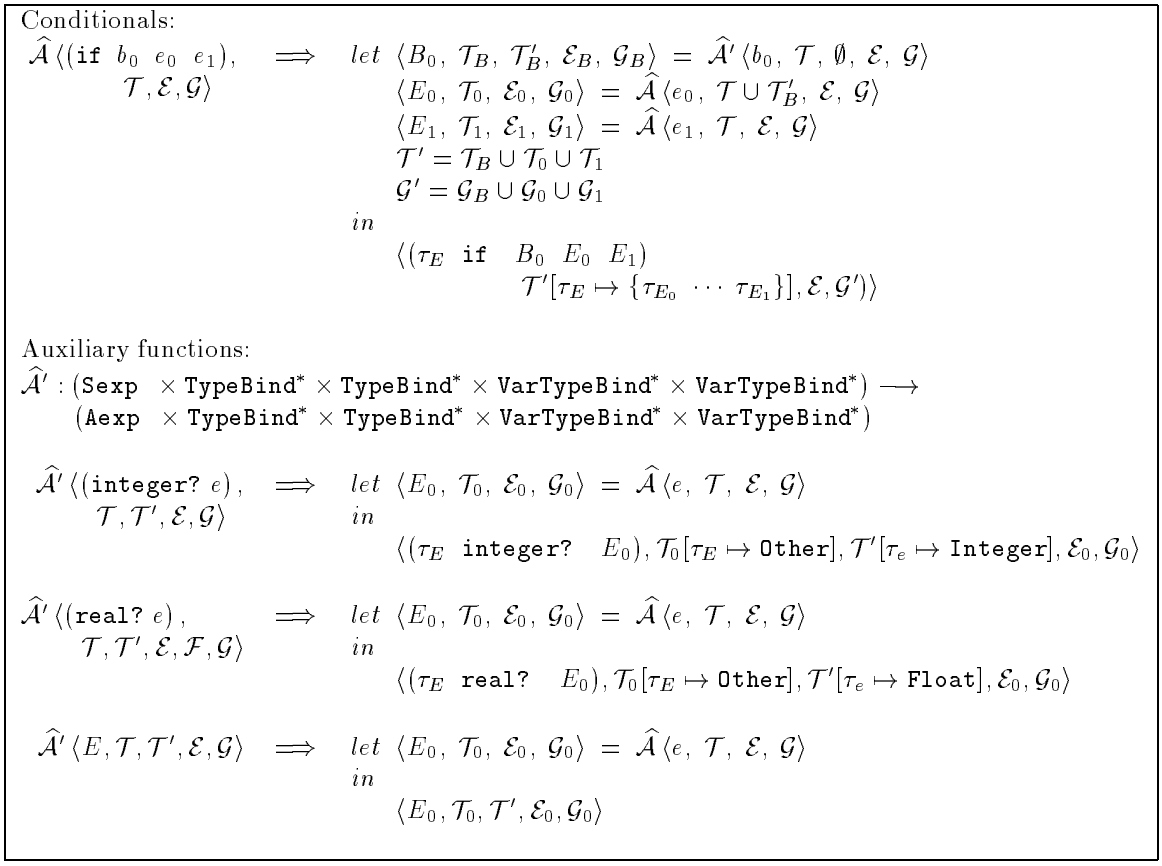

Figure 7: Annotation algorithm for conditionals.

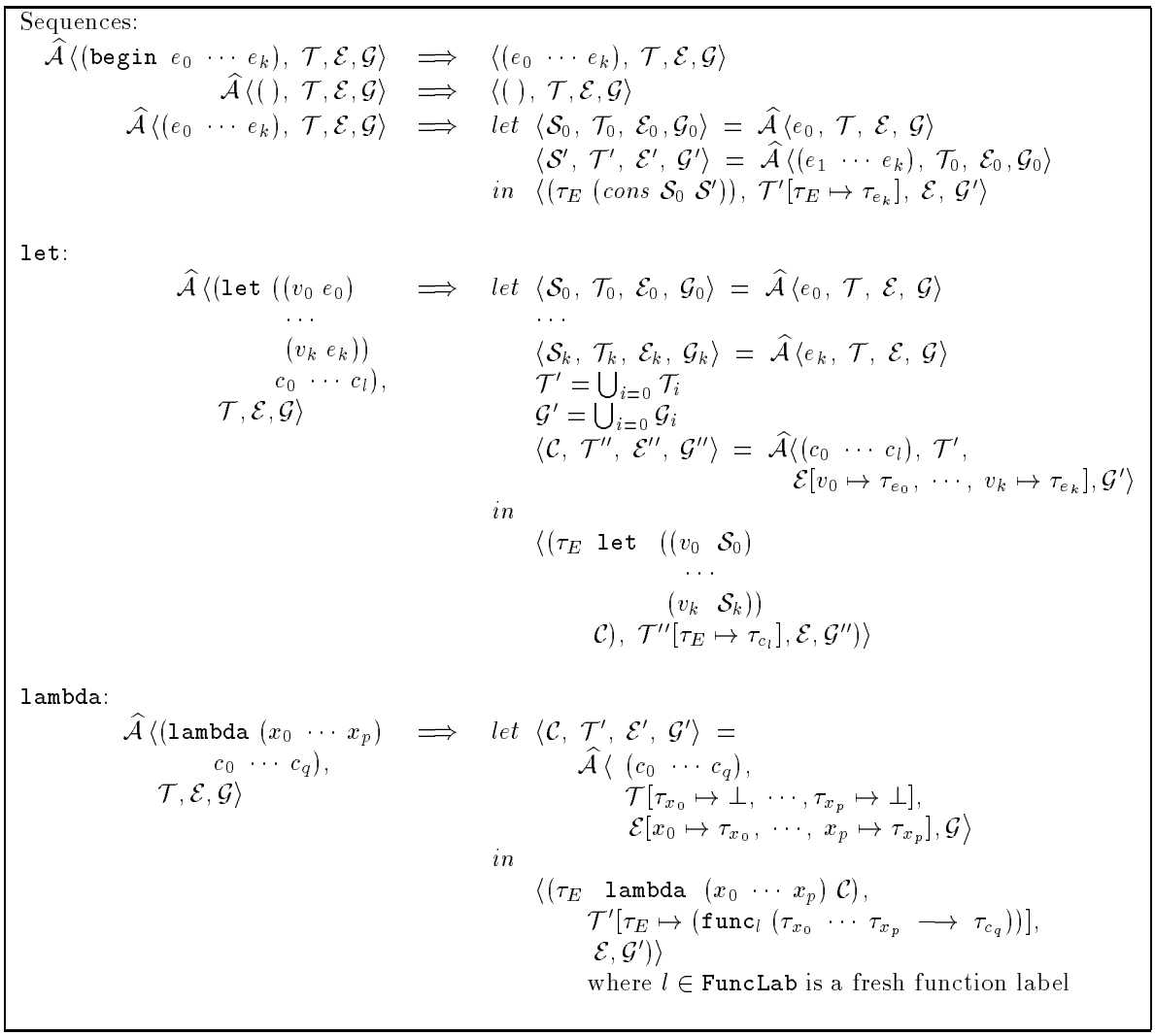

Figure 8: Annotation algorithm for sequence, `let` and `lambda`.

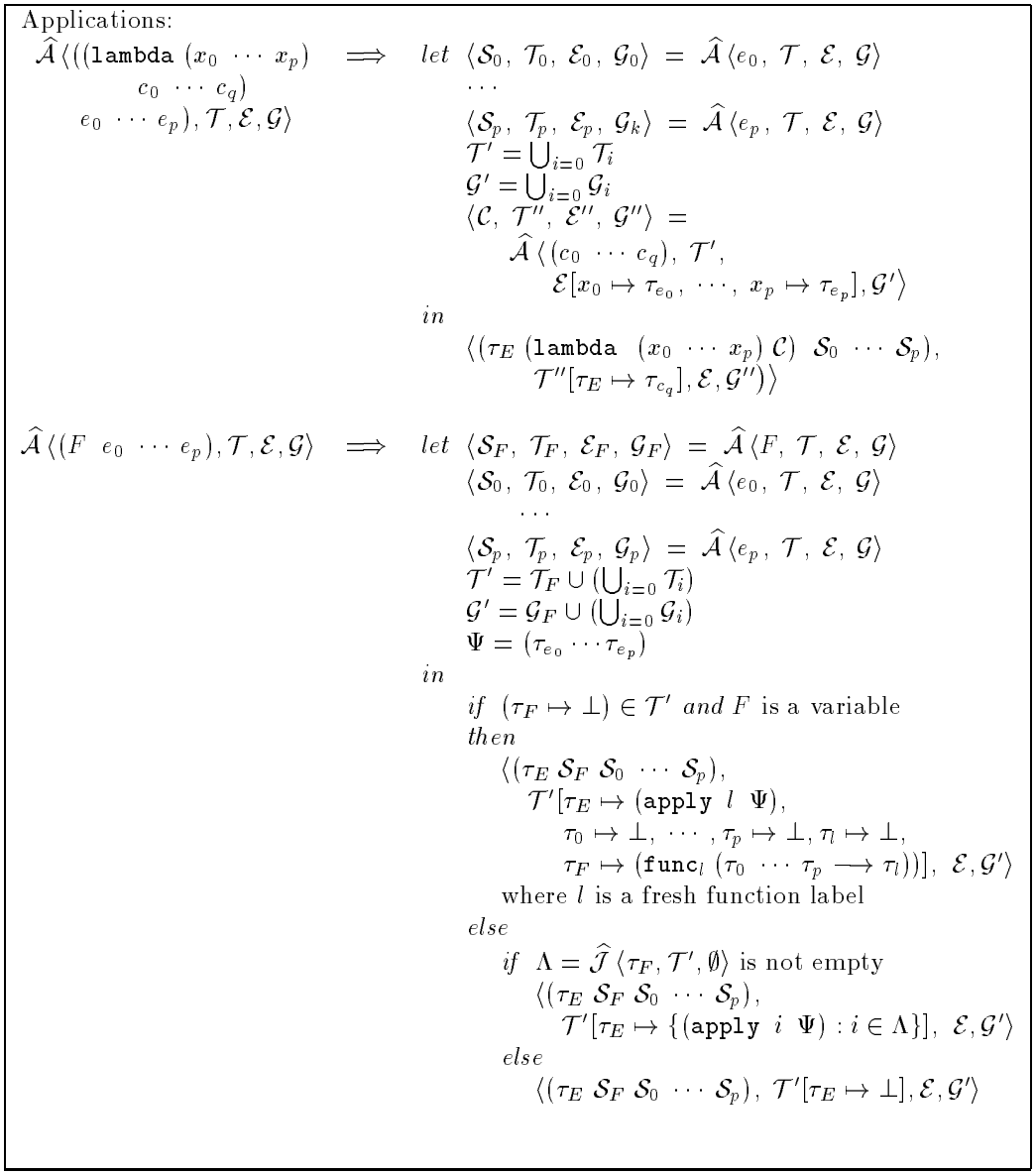

Figure 9: Annotation algorithm for function application.

$$\widehat{\mathcal{J}} : \{\texttt{SimpleType} \mid \texttt{TypeSeq}\} \times \texttt{TypeBind}^* \times \texttt{TypeVar}^* \longrightarrow \texttt{FuncType}^*$$

$$\widehat{\mathcal{J}}\,\langle t, \mathcal{T}, \mathcal{M}\rangle = \emptyset \qquad \text{if } t \in \texttt{SimpleType}$$

$$\widehat{\mathcal{J}}\,\langle \tau, \mathcal{T}, \mathcal{M}\rangle = \widehat{\mathcal{J}}\,\langle \tau_0, \mathcal{T}, \mathcal{M} \cup \{\tau\}\rangle \qquad \text{if } [\tau \mapsto \tau_0] \in \mathcal{T} \text{ and } \tau_0 \in \texttt{TypeVar}$$

$$\widehat{\mathcal{J}}\,\langle \tau, \mathcal{T}, \mathcal{M}\rangle = \emptyset \qquad \text{if } \tau \in \mathcal{M}$$

$$\widehat{\mathcal{J}}\,\langle \{\tau_0, \cdots, \tau_k\}, \mathcal{T}, \mathcal{M}\rangle = \bigcup_i \widehat{\mathcal{J}}\,\langle \tau_i, \mathcal{T}, \mathcal{M} \cup \{\tau_0, \cdots, \tau_{i-1}, \tau_{i+1}, \cdots, \tau_k\}\rangle$$

$$\widehat{\mathcal{J}}\,\langle (\texttt{op}\ \tau_0, \cdots, \tau_k), \mathcal{T}, \mathcal{M}\rangle = \textit{let } t_0 = \widehat{\mathcal{J}}\,\langle \tau_0, \mathcal{T}, \mathcal{M} \cup \{\tau_1, \cdots, \tau_k\}\rangle \quad \cdots \quad t_i = \widehat{\mathcal{J}}\,\langle \tau_i, \mathcal{T}, \mathcal{M} \cup \{\tau_0, \cdots, \tau_{i-1}, \tau_{i+1}, \cdots, \tau_k\}\rangle \quad \cdots \quad t_k = \widehat{\mathcal{J}}\,\langle \tau_k, \mathcal{T}, \mathcal{M} \cup \{\tau_0, \cdots, \tau_{k-1}\}\rangle \textit{ in } \textit{if } (\texttt{op}\ t_0 \cdots t_k) \textit{ is defined then return the result of evaluating } (\texttt{op}\ t_0 \cdots t_k) \textit{ else } \emptyset \qquad \text{where } \texttt{op} \text{ is a list operation.}$$

$$\widehat{\mathcal{J}}\,\langle (\texttt{apply}\ l\ \Psi), \mathcal{T}, \mathcal{M}\rangle = \widehat{\mathcal{J}}\,\langle \tau, \mathcal{T}, \mathcal{M}\rangle \qquad \text{where } (\texttt{func}_l(\Upsilon \longrightarrow \tau)) \in \mathcal{T}$$

$$\widehat{\mathcal{J}}\,\langle (\texttt{call/cc}\ l), \mathcal{T}, \mathcal{M}\rangle = \widehat{\mathcal{J}}\,\langle \bigcup_i \Psi_{l_i}, \mathcal{T}, \mathcal{M}\rangle \qquad \forall (\texttt{apply}\ l\ \Psi_{l_i}) \in \mathcal{T}$$

$$\widehat{\mathcal{J}}\,\langle (\texttt{func}_l(\Upsilon \longrightarrow \tau_l)), \mathcal{T}, \mathcal{M}\rangle = \{l\}$$

Figure 10: Definition of $\widehat{\mathcal{J}}$.

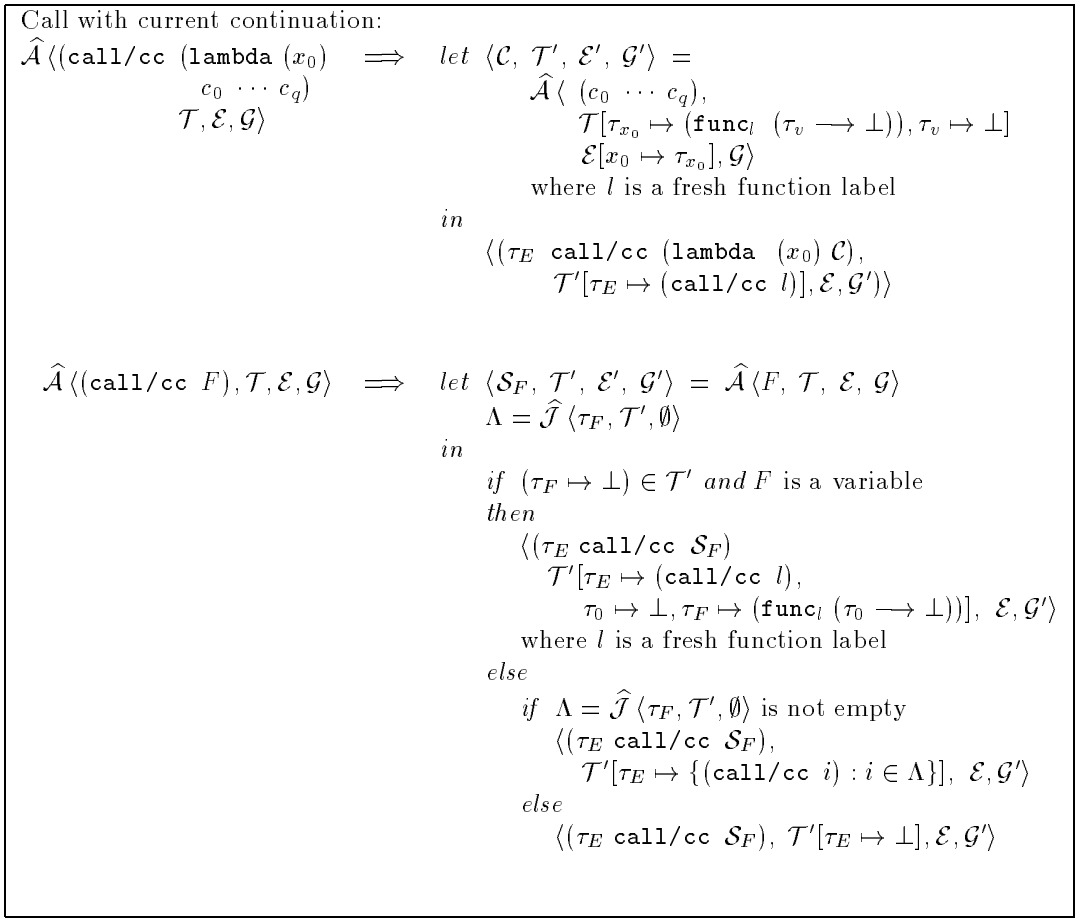

Figure 11: Annotation algorithm for continuation.

$\widehat{\mathcal{I}} : \{\texttt{SimpleType} \mid \texttt{TypeSeq}\} \times \texttt{TypeBind}^* \times \texttt{TypeVar}^* \longrightarrow \texttt{SimpleType}^*$

$$\widehat{\mathcal{I}}\,\langle t, \mathcal{T}, \mathcal{M}\rangle = t \qquad \text{if } t \in \texttt{SimpleType}$$

$$\widehat{\mathcal{I}}\,\langle \tau, \mathcal{T}, \mathcal{M}\rangle = t \qquad \text{if } [\tau \mapsto t] \in \mathcal{T} \text{ and } t \in \texttt{SimpleType}$$

$$\widehat{\mathcal{I}}\,\langle \tau, \mathcal{T}, \mathcal{M}\rangle = \widehat{\mathcal{I}}\,\langle \tau_0, \mathcal{T}, \mathcal{M} \cup \{\tau\}\rangle \qquad \text{if } [\tau \mapsto \tau_0] \in \mathcal{T} \text{ and } \tau_0 \in \texttt{TypeVar}$$

$$\widehat{\mathcal{I}}\,\langle \tau, \mathcal{T}, \mathcal{M}\rangle = \bot \qquad \text{if } \tau \in \mathcal{M}$$

$$\widehat{\mathcal{I}}\,\langle \{\tau_0, \cdots, \tau_k\}, \mathcal{T}, \mathcal{M}\rangle = \textit{let } t_0 = \widehat{\mathcal{I}}\,\langle \tau_0, \mathcal{T}, \mathcal{M} \cup \{\tau_1, \cdots, \tau_k\}\rangle \cdots t_i = \widehat{\mathcal{I}}\,\langle \tau_i, \mathcal{T}, \mathcal{M} \cup \{\tau_0, \cdots, \tau_{i-1}, \tau_{i+1}, \cdots, \tau_k\}\rangle \cdots t_k = \widehat{\mathcal{I}}\,\langle \tau_k, \mathcal{T}, \mathcal{M} \cup \{\tau_0, \cdots, \tau_{k-1}\}\rangle \textit{ in } \bigsqcup_i t_i$$

where $\bigsqcup_i$ is the *least upper bound* of the simple types t_i in the type hierarchy defined in Fig. 2.

$$\widehat{\mathcal{I}}\,\langle (\texttt{op}\ \tau_0, \cdots, \tau_k), \mathcal{T}, \mathcal{M}\rangle = \textit{let } t_0 = \widehat{\mathcal{I}}\,\langle \tau_0, \mathcal{T}, \mathcal{M} \cup \{\tau_1, \cdots, \tau_k\}\rangle \cdots t_i = \widehat{\mathcal{I}}\,\langle \tau_i, \mathcal{T}, \mathcal{M} \cup \{\tau_0, \cdots, \tau_{i-1}, \tau_{i+1}, \cdots, \tau_k\}\rangle \cdots t_k = \widehat{\mathcal{I}}\,\langle \tau_k, \mathcal{T}, \mathcal{M} \cup \{\tau_0, \cdots, \tau_{k-1}\}\rangle \textit{ in } \textit{if } (\texttt{op}\ t_0 \cdots t_k) \textit{ is defined then return the result of evaluating } (\texttt{op}\ t_0 \cdots t_k) \textit{ else } \bot$$

where op is a list operation.

$$\widehat{\mathcal{I}}\,\langle (\texttt{func}_l\ (\Upsilon \longrightarrow \tau)), \mathcal{T}, \mathcal{M}\rangle = \texttt{Other}$$

$$\widehat{\mathcal{I}}\,\langle (\texttt{apply}\ l\ \Psi), \mathcal{T}, \mathcal{M}\rangle = \widehat{\mathcal{I}}\,\langle \tau, \mathcal{T}, \mathcal{M}\rangle \qquad \text{where } (\texttt{func}_l(\Upsilon \longrightarrow \tau)) \in \mathcal{T}$$

$$\widehat{\mathcal{I}}\,\langle (\texttt{call/cc}\ l), \mathcal{T}, \mathcal{M}\rangle = \widehat{\mathcal{I}}\,\langle \bigcup_i \Psi_{l_i}, \mathcal{T}, \mathcal{M}\rangle \qquad \forall (\texttt{apply}\ l\ \Psi_{l_i}) \in \mathcal{T}$$

Figure 12: Definition of $\widehat{\mathcal{I}}$.

ALGORITHM SPECIALIZEFUNCTION

INPUTS: – $\langle \mathcal{S}_\mathcal{P}, \mathcal{T}, \mathcal{E}, \mathcal{G} \rangle = \widehat{\mathcal{A}} \langle \mathcal{P}, \emptyset, \emptyset, \emptyset \rangle$
– $\mathcal{R}$ initially empty.

OUTPUTS: $\mathcal{S}_\mathcal{P}$ - the annotated program with specialized functions added;
$\mathcal{T}$ - the new set of type bindings;
$\mathcal{G}$ - the new set of global variable bindings;
$\mathcal{R}$ - the function rename set;

1. For each function label, l say, do step 2.

2. Compute $\Psi = \bigcup_i \Psi_{l_i}$ where $(\texttt{apply}\ \ l\ \ \Psi_{l_i}) \in \mathcal{T}$.

3. Partition Ψ into sets $\psi_0, \cdots, \psi_n$ such that
$$\forall (\tau_{e_0}\ \cdots\ \tau_{e_k}), (\tau_{f_0}\ \cdots\ \tau_{f_k}) \in \psi_i : \widehat{\mathcal{I}} \left\langle \tau_{e_j}, \mathcal{T}, \emptyset \right\rangle = \widehat{\mathcal{I}} \left\langle \tau_{f_j}, \mathcal{T}, \emptyset \right\rangle$$

4. If there is only one such partition
4.1 then pick a tuple in Ψ, $(\tau_{e_0}\ \cdots\ \tau_{e_k})$ say and update $\mathcal{T}$ as follows:
$$\mathcal{T} \longleftarrow \mathcal{T}[\tau_0 \mapsto \tau_{e_0},\ \cdots,\ \tau_k \mapsto \tau_{e_k}]$$

4.2 else for each ψ_i do

4.2.1 Pick a tuple in ψ_i, $(\tau_{e_0}\ \cdots\ \tau_{e_k})$ say;

4.2.2 Obtain the body of F, $E = (\texttt{lambda}\ (x_0\ \cdots\ x_k)\ \ c_0\ \cdots\ c_v)$ say, from $\mathcal{P}$;

4.2.3 Pick a name F_i say, such that $(F_i \mapsto \tau) \notin \mathcal{G}$, for some τ, i.e. it has not been used before;

4.2.4 Let $\langle \mathcal{S}_{F_i}, \mathcal{T}_{F_i}, \mathcal{E}_{F_i}, \mathcal{G}_{F_i} \rangle =$
$$\widehat{\mathcal{A}} \langle (c_0\ \cdots\ c_v), \mathcal{T}, \mathcal{E}[x_0 \mapsto \tau_{e_0},\ \cdots,\ x_k \mapsto \tau_{e_k}], \mathcal{G} \rangle$$

4.2.5 Update as follows:
a) $\mathcal{S}_\mathcal{P} \longleftarrow \mathcal{S}_\mathcal{P} \cup \{(\tau_{s_i}\ \texttt{set!}\ F_i\ (\tau_{F_i}\ \texttt{lambda}\ (x_0\ \cdots\ x_k) \mathcal{S}_{F_i})\}$
b) $\mathcal{T} \longleftarrow \mathcal{T}_{F_i}[\tau_{s_i} \mapsto \bot, \tau_{F_i} \mapsto (\texttt{func}\ (\tau_{e_0}\ \cdots\ \tau_{e_k}\ \longrightarrow\ \tau_{c_v}))]$
c) $\mathcal{G} \longleftarrow \mathcal{G}_{F_i}[F_i \mapsto \tau_{F_i}]$
d) $\mathcal{R} \longleftarrow \mathcal{R} \cup \{(F, F_i, (\tau_{e_0}\ \cdots\ \tau_{e_k}\ \longrightarrow\ \tau_{c_l}))\}$

Figure 13: Specialization of function calls.

$\widehat{\mathcal{D}} : (\texttt{Aexp} \times \texttt{TypeBind}^* \times \texttt{VarTypeBind}^* \times \texttt{VarTypeBind}^* \times \texttt{RenameSet}) \longrightarrow$
$(\texttt{Sexp} \times \texttt{TypeBind}^* \times \texttt{VarTypeBind}^* \times \texttt{VarTypeBind}^* \times \texttt{RenameSet})$

Arithmetic Operators:

$$
\begin{array}{l}
\widehat{\mathcal{D}}\,\langle(\tau_E\ \texttt{op}\ e_0\ e_1), \mathcal{T}, \mathcal{E}, \mathcal{G}, \mathcal{R}\rangle \implies \mathit{let}\ \langle \mathcal{S}_0,\ \mathcal{T}_0,\ \mathcal{E}_0,\ \mathcal{G}_0,\ \mathcal{R}\rangle = \widehat{\mathcal{D}}\,\langle e_0,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G},\ \mathcal{R}\rangle \\
\qquad \langle \mathcal{S}_1,\ \mathcal{T}_1,\ \mathcal{E}_1,\ \mathcal{G}_1,\ \mathcal{R}\rangle = \widehat{\mathcal{D}}\,\langle e_1,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G},\ \mathcal{R}\rangle \\
\qquad \mathcal{T}' = \mathcal{T}_0 \cup \mathcal{T}_1 \\
\qquad \mathcal{G}' = \mathcal{G}_0 \cup \mathcal{G}_1 \\
\qquad \mathit{in} \\
\qquad\quad \mathit{if}\ \widehat{\mathcal{I}}\,\langle \tau_{e_0}, \mathcal{T}', \emptyset\rangle = \texttt{Integer}\ \mathit{and}\ \widehat{\mathcal{I}}\,\langle \tau_{e_1}, \mathcal{T}', \emptyset\rangle = \texttt{Integer} \\
\qquad\quad \mathit{then} \\
\qquad\qquad \langle(\texttt{int-op}\ \mathcal{S}_0\ \ \mathcal{S}_1), \mathcal{T}', \mathcal{E}[\tau_E \mapsto \texttt{Integer}], \mathcal{G}', \mathcal{R}\rangle \\
\qquad\quad \mathit{else} \\
\qquad\qquad \mathit{if}\ \widehat{\mathcal{I}}\,\langle \tau_{e_0}, \mathcal{T}', \emptyset\rangle = \texttt{Float} \\
\qquad\qquad \mathit{then} \\
\qquad\qquad\quad \mathit{if}\ \widehat{\mathcal{I}}\,\langle \tau_{e_1}, \mathcal{T}', \emptyset\rangle = \texttt{Float} \\
\qquad\qquad\quad \mathit{then} \\
\qquad\qquad\qquad \langle(\texttt{fp-op}\ \mathcal{S}_0\ \ \mathcal{S}_1), \mathcal{T}', \mathcal{E}[\tau_E \mapsto \texttt{Float}], \mathcal{G}', \mathcal{R}\rangle \\
\qquad\qquad\quad \mathit{else} \\
\qquad\qquad\qquad \langle(\texttt{fp-op}\ \mathcal{S}_0\ \ (\texttt{exact->inexact}\ \mathcal{S}_1)), \\
\qquad\qquad\qquad\quad \mathcal{T}', \mathcal{E}[\tau_E \mapsto \texttt{Float}], \mathcal{G}', \mathcal{R}\rangle \\
\qquad\qquad \mathit{else} \\
\qquad\qquad\quad \mathit{if}\ \widehat{\mathcal{I}}\,\langle \tau_{e_1}, \mathcal{T}', \emptyset\rangle = \ \texttt{Float} \\
\qquad\qquad\quad \mathit{then} \\
\qquad\qquad\qquad \langle(\texttt{fp-op}\ (\texttt{exact->inexact}\ \mathcal{S}_0)\ \ \mathcal{S}_1), \\
\qquad\qquad\qquad\quad \mathcal{T}', \mathcal{E}[\tau_E \mapsto \texttt{Float}], \mathcal{G}', \mathcal{R}\rangle \\
\qquad\qquad\quad \mathit{else} \\
\qquad\qquad\qquad \langle(\texttt{op}\ \mathcal{S}_0\ \ \mathcal{S}_1), \mathcal{T}', \mathcal{E}, \mathcal{G}', \mathcal{R}\rangle \\
\qquad \text{where } \texttt{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}
\end{array}
$$

Application:

$$
\begin{array}{l}
\widehat{\mathcal{D}}\,\langle(\tau_F\ \ F\ \ e_0\ \cdots\ e_p), \mathcal{T}, \mathcal{E}, \mathcal{G}, \mathcal{R}\rangle \implies \mathit{let}\ \langle \mathcal{S}_0,\ \mathcal{T}_0,\ \mathcal{E}_0,\ \mathcal{G}_0,\ \mathcal{R}\rangle = \widehat{\mathcal{D}}\,\langle e_0,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G},\ \mathcal{R}\rangle \\
\qquad \cdots \\
\qquad \langle \mathcal{S}_p,\ \mathcal{T}_p,\ \mathcal{E}_p,\ \mathcal{G}_p,\ \mathcal{R}\rangle = \widehat{\mathcal{D}}\,\langle e_p,\ \mathcal{T},\ \mathcal{E},\ \mathcal{G},\ \mathcal{R}\rangle \\
\qquad \mathcal{T}' = \bigcup_{i=0} \mathcal{T}_i \\
\qquad \mathcal{G}' = \bigcup_{i=0} \mathcal{G}_i \\
\qquad \mathit{in} \\
\qquad\quad \mathit{if}\ \exists F_i,\ \tau_0,\ \cdots \tau_k,\ \tau_c : \\
\qquad\qquad\quad (F, F_i, (\tau_0\ \cdots\ \tau_k\ \longrightarrow\ \tau_c)) \in \mathcal{R}\ \mathit{and} \\
\qquad\qquad\quad \widehat{\mathcal{I}}\,\langle \tau_{e_0}, \mathcal{T}', \emptyset\rangle = \widehat{\mathcal{I}}\,\langle \tau_0, \mathcal{T}'\rangle\ \mathit{and} \\
\qquad\qquad\qquad\qquad \cdots \\
\qquad\qquad\quad \widehat{\mathcal{I}}\,\langle \tau_{e_k}, \mathcal{T}', \emptyset\rangle = \widehat{\mathcal{I}}\,\langle \tau_k, \mathcal{T}', \emptyset\rangle \\
\qquad\quad \mathit{then} \\
\qquad\qquad \langle(F_i\ \mathcal{S}_0\ \cdots\ \mathcal{S}_k), \mathcal{T}'[\tau_E \mapsto \tau_c], \mathcal{E}, \mathcal{G}', \mathcal{R}\rangle \\
\qquad\quad \mathit{else} \\
\qquad\qquad \langle(F\ \mathcal{S}_0\ \cdots\ \mathcal{S}_k), \mathcal{T}', \mathcal{E}, \mathcal{G}', \mathcal{R}\rangle
\end{array}
$$

Figure 14: Program Transformation for Arithmetic Expressions and Function Calls.

```
(define (+-TWO x y)
    (cond ((fixed? x)
           (cond ((fixed? y)
                  ((lap (x y) (_TSCP (IPLUS (_S2CINT x) (_S2CINT y)))) x y))
                 ((float? y)
                  ((lap (x y) (FLTV_FLT (PLUS (FIX_FLTV x) (FLOAT_VALUE y))))
                   x y))
                 (else (error '+ "Argument not a NUMBER: ~s" y))))
          ((fixed? y)
           (cond ((float? x)
                  ((lap (x y) (FLTV_FLT (PLUS (FLOAT_VALUE x) (FIX_FLTV y))))
                   x y))
                 (else (error '+ "Argument not a NUMBER: ~s" x))))
          ((and (float? x) (float? y))
           ((lap (x y) (FLTV_FLT (PLUS (FLOAT_VALUE x) (FLOAT_VALUE y))))
            x y))
          (else (error '+ "Argument(s) not a NUMBER: ~s ~s" x y))))
```

Figure 15: Implementation of addition in Scheme→C.

Benchmark	**Original** (# of GC calls)	**Opt-1/Impr.** (# of GC calls)	**Opt-2/Impr.** (#GC/Impr.)
`Linpack`			
$n = 100$	288.11 kflops (14)	332.25 kflops / 15.32% (14)	535.06 kflops / 85.71% (5 / 64.28 %)
$n = 200$	204.92 kflops (60)	226.97 kflops / 10.76% (71)	412.71 kflops / 101.40% (38 / 36.67%)
$n = 300$	157.84 kflops (164)	172.21 kflops / 9.10% (164)	318.77 kflops / 101.95% (84 / 48.78%)
$n = 400$	*crashed* (283†)	*crashed* (—)	184.61 kflops / ∞ (173 / —)
mp3d (3000 molecules)			
first 100 steps	46.37 secs (32)	35.18 secs / 24.13% (32)	34.87 secs / 24.8% (31 / 3.13%)
next 100 steps	59.98 secs (43)	49.50 secs / 17.47% (35)	50.05 secs / 16.22% (42 / 2.32%)
next 100 steps	70.17 secs (39)	58.80 secs / 16.20% (38)	54.77 secs / 21.95% (38 / 2.56%)
next 100 steps	70.63 secs (45)	59.60 secs / 15.61% (44)	59.30 secs / 16.04% (42 / 6.67%)
Simplex			
10000 repetitions	49.62 secs (246)	46.85 secs / 5.6% (254)	41.53 secs / 16.36% (219 / 11.78%)

Table 1: Performance of Type Inference System.