1 Introduction

Scheme is a *latently typed* language [R3RS]. This means that unlike statically typed languages such as ML or Pascal, types are associated only with run-time values, not with variables. A variable can potentially be bound to values of any type, and type tests can be performed at run time to determine the execution path through a program. This gives the programmer an extremely powerful form of polymorphism without requiring a verbose system of type declarations.

This power and convenience is not without cost, however. Since run-time behavior can determine which values are bound to a given variable, precisely determining the type of a given reference to a variable is undecidable at compile time. For example, consider the following fragment of Scheme code:

```
(let ((x (if (oracle) 3 'three)))
  (f x))
```

The reference to `x` in the subexpression `(f x)` could have any one of the types integer, symbol, integer+symbol, or bottom, depending on whether the oracle always returns true, always returns false, sometimes returns both, or never returns at all, respectively. Of course, determining the oracle's behavior at compile time is, in general, undecidable.

We can appeal to data-flow analysis techniques to recover a conservative approximation of the type information implicit in a program's structure. However, Scheme is a difficult language to flow analyse: its higher-order procedures and first-class continuations render the construction of a control-flow graph at compile time very difficult. The problem of conservatively approximating, at compile time, the types of the values associated with the references to variables in a Scheme program thus involves data-flow analysis in the presence of higher-order functions.

In this paper, I present an algorithm for flow analysis that can correctly recover a useful amount of type information from Scheme programs, even in the presence of higher-order functions and `call/cc`. This algorithm performs a kind of type inference. Since the term "type inference" is commonly used to refer to the recovery of static type assignments for variables, I will instead refer to the type analysis performed by this algorithm as *type recovery*: the recovery of type information from the control and environment structure of a program.

1.1 Sources of Type Information

Type information in Scheme programs can be statically recovered from three sources: conditional branches, applications of primitive operations, and user declarations.

1.1.1 Conditional Branches

Consider a simple version of the Scheme `equal?` predicate:

```
(define (equal? a b)
  (cond ((number? a)
         (and (number? b) (= a b)))
        ((pair? a)
         (and (pair? b)
              (equal? (car a) (car b))
              (equal? (cdr a) (cdr b))))
        (else (eq? a b))))
```

There are three arms in the `cond` form. References to `a` down the first arm are guaranteed to have type number. Furthermore, in the numeric comparison form, `(= a b)`, an assumption can be made that `b` is also a number, since the `AND` guarantees that control reaches the comparison only if `(number? b)` is true. Similarly, we can infer that references to `a` in the second arm of the `cond` have the pair type and, less usefully, that references to `a` in the third arm of the `cond` do not have either type pair or number. It is important to realise from this example that Scheme type recovery assigns types not to variables, but to variable *references*. The references to `a` in the different arms of the `cond` all have different types.

Type recovery of this sort can be helpful to a Scheme compiler. If the compiler can determine that the `(= a b)` form is guaranteed to compare only numbers, it can compile the comparison without a run-time type check, gaining speed without sacrificing safety. Similarly, determining that the `(car a)`, `(car b)`, `(cdr a)`, and `(cdr b)` forms are all guaranteed to operate on pairs allows the compiler to safely compile the `car` and `cdr` applications without run-time type checks.

1.1.2 Primop Application

An obvious source of type information is the application of primitive operations (called “primops”) such as `+` and `cons`. Clearly the result of `(cons a b)` is a pair. In addition, we can recover information about the type of subsequent references to primop arguments. For example, after the primop application `(cdr p)`, references to `p` along the same control path can be assumed to have the pair type. {Note Recovering Primops}

As a simple example, consider the following Scheme expression:

```
(let* ((a (cdr b))
       (q (char->integer (vector-ref a i))))
  ... (car b) ... (+ q 2))
```

References to `b` occurring after the `(cdr b)` form are guaranteed to have the pair type — otherwise, the `cdr` application would have rendered the effect of the computation to be undefined. Hence, the subsequent `(car b)` application does not need to do a type check. Similarly, after evaluating the second clause of the `let*`, references to `a` have type vector and references to `i` and `q` are small integers, because `vector-ref` requires an integer index, and `char->integer` generates an integer result. Hence the compiler can omit all type checks in the object code for `(+ q 2)`, and simply open code the addition.

1.1.3 User Declarations

Type inference can also pick up information from judiciously placed declarations inserted by the programmer. There are essentially two types of user declarations, one requesting a run-time type check to enforce the verity of the declaration, and one simply asserting a type at compile time, in effect telling the compiler, "Trust me. This quantity will always have this type. Assume it; don't check for it."

The first kind of declaration is just T's `enforce` procedure [T], which can be defined as:

```
(define (enforce pred val) (if (pred val) val (error)))
```

`Enforce` simply forces a run-time type check at a point the user believes might be beneficial to the compile-time analysis, halting the program if the check is violated. For example,

`(block (enforce number? x)` ...*forms*...`)`

allows the compiler to assume that references to `x` in the forms following the `enforce` have the number type. `Enforce` allows execution to proceed into the body of

`(let ((y (enforce integer? (foo 3)))) ` *body*`)`

only if `(foo 3)` returns an integer. Clearly, `enforce`-based type recovery is simply conditional-branch based type recovery.

The "trust me" sort of user declaration is expressed in Scheme via the `proclaim` procedure, which asserts that its second argument has the type specified by its first argument. For example, `(proclaim symbol? y)` tells the compiler to believe that `y` has type symbol. Like `enforce`, `proclaim` can also be viewed as a kind of conditional expression:

```
(define (proclaim pred val) (if (pred val) val ($)))
```

where `($)` denotes a computation with undefined effect. Since an undefined effect means that "anything goes," the compiler is permitted to elide the conditional expression altogether and simply take note of the programmer's assertion that `val` has the declared type. Incorrect assertions will still result in undefined effects.

1.2 Type Recovery from Multiple Sources

All three sources of type information — conditional branches, primop applications, and user declarations — can be used together in recovering type information from programs, thereby enabling many optimizations. Consider the `delq` procedure of Figure 1. Because `ans` is only bound to the constant `'()`, itself, and the result of a `cons` application, it must always be a list. So all references to `ans` are completely typeable at compile time. Because of the conditional type test `(pair? rest)`, `car` and `cdr` are guaranteed to be applied to legitimate pair values. Thus compile-time type recovery can guarantee the full type safety of `delq` with no extra run-time type checks.

For a numerical example, consider the factorial procedure in Figure 1. Note the use of an explicit run-time type check, `(enforce integer? n)`, to force the subsequent reference to `n` to be of integer type. With the help of this user declaration, the analysis can determine that `m` is always bound to an integer, and therefore, that `ans` must also be bound to an integer. Thus, the factorial function written with generic arithmetic operators can have guaranteed type safety for the primop applications and also specialise the generic arithmetic operations to integer operations, at the expense of a single run-time type check per `fact` call.

```
(define (delq elt lis)
  (letrec ((lp (λ (ans rest)
                 (if (pair? rest)
                     (let ((head (car rest)))
                       (lp (if (eq? head elt) ans
                               (cons head ans))
                           (cdr rest)))
                     (reverse! ans)))))
    (lp '() lis)))

(define (fact n)
  (letrec ((lp (λ (ans m)
                 (if (= m 0) ans
                     (lp (* ans m) (- m 1))))))
    (enforce integer? n)
    (lp 1 n)))
```

Figure 1: Scheme `delq` and factorial

If we eliminate the `enforce` form, then the type recovery can do less, because `fact` could be called on bogus, non-integer values. However, if the equality primop `(= m 0)` requires its arguments to have the same type, we can infer that references to `m` after the equality test must be integer references, and so the multiplication and subtraction operations are guaranteed to be on integer values. Hence, even in the naive, declaration-free case, type-recovery analysis is able to pick up enough information from the code to guarantee the type safety of `fact` with only a single type check per iteration, as opposed to the four type checks required in the absence of any analysis.

The implementation of the type recovery algorithm discussed in Section 6 can, in fact, recover enough information from the `delq` and `fact` procedures of figure 1 to completely assign precise types to all variable references, as discussed above. For these examples, at least, compile-time type analysis can provide run-time safety with no execution penalty.

1.3 Overview of the Paper

The remainder of this paper will describe some of the details of the type recovery algorithm. Section 2 introduces the notion of quantity-based analysis, which underlies the type recovery algorithm. Section 3 briefly reviews CPS Scheme, the intermediate representation used by the analysis, and the non-standard abstract semantic (NSAS) interpretation approach to program analysis that is the general framework for the type recovery analysis. Section 4 then uses the quantity model and the NSAS framework to present a “perfect” (and hence uncomputable) type recovery analysis for CPS Scheme. Section 5 abstracts the perfect analysis to a computable, useful approximate type recovery analysis. Section 6 describes an implementation of the approximate type recovery algorithm. Section 7 is a collection of assorted discussions and speculations on Scheme type recovery. Finally, Section 8 discusses related work.

2 Quantity-based Analysis

Type recovery is an example of what I call a *quantity-based analysis*. Consider the Scheme expression

```
(if (< j 0) (- j) j)
```

Whatever number `j` names, we know that it is negative in the then-arm of the `if` expression, and non-negative in the else-arm. In short, we are associating an abstract property of `j`'s value (its sign) with control points in the program. It is important to realize that we are tracking information about *quantities* (`j`'s value), not *variables* (`j` itself). For example, consider the following expression:

```
(let ((i j)) (if (< j 0) (foo i)))
```

Clearly, the test involving `j` determines information about the value of `i` since they *both name the same quantity*. In a quantity-based analysis, information is tracked for quantities, and quantities can be named by multiple variables. This information can then be associated with the variable references appearing in the program. For the purposes of keeping track of this information, we need names for quantities; variables can then be bound to these quantity names (which are called *qnames*), which in turn have associated abstract properties.

In principle, calls to primops such as `+` or `cons` create new qnames since these operations involve the creation of new computational quantities. On the other hand, lambda binding simply involves the binding of a variable to an already existing qname. In practice, extra qnames often must be introduced since it can be difficult to determine at compile-time which qname a variable is bound to. Consider, for example, the following procedure:

```
(define (foo x y) (if (integer? x) y 3))
```

It might be the case that all calls to `foo` are of the form `(foo a a)`, in which case `x` and `y` can refer to the same qname. But if the analysis cannot determine this fact at compile time, `x` and `y` must be allocated two distinct qnames; hence determining information about `x`'s value will not shed light on `y`'s value.

As another example, consider the vector reference in the expression:

```
(let ((y (vector-ref vec i))) ...)
```

Now, `y` is certainly bound to an existing quantity, but it is unlikely that a compile-time analysis will be able to determine which one. So, a new qname must be assigned to `y`'s binding.

In general, then, a conservative, computable, quantity-based analysis approximates the tracking of information on run-time values by introducing new qnames whenever it is unable to determine to which existing qname a variable is bound. These extra qnames limit the precision of the analysis, but force the analysis to err conservatively.

3 CPS and NSAS

3.1 CPS

The intermediate representation used for type recovery is Continuation-Passing Style Scheme, or CPS Scheme. CPS Scheme is a simple variant of Scheme in which procedures do not return, side-effects are allowed on data structures but not on variables, and all transfers of control — sequencing, conditional branching, loops, and subroutine call/return — are represented by procedure calls. This simple language is a surprisingly useful intermediate representation: variants of CPS Scheme have been used as the intermediate representation for several Scheme, Common Lisp and ML compilers [Rabbit] [ORBIT] [MLComp].

CPS Scheme has the following simple syntax:

$$\begin{array}{rcll}
\text{PR} & ::= & \text{LAM} & \\
\text{LAM} & ::= & (\lambda\ (v_1 \ldots v_n)\ c) & [v_i \in \text{VAR},\ c \in \text{CALL}] \\
\text{CALL} & ::= & (f\ \ a_1 \ldots a_n) & [f \in \text{FUN},\ a_i \in \text{ARG}] \\
 & & (\texttt{letrec}\ ((f_1\ \ l_1)\ldots)\ c) & [f_i \in \text{VAR},\ l_i \in \text{LAM},\ c \in \text{CALL}] \\
\text{FUN} & ::= & \text{LAM} + \text{REF} + \text{PRIM} & \\
\text{ARG} & ::= & \text{LAM} + \text{REF} + \text{CONST} & \\
\text{REF} & ::= & \text{VAR} & \\
\text{VAR} & ::= & \{\texttt{x}, \texttt{z}, \texttt{foo}, \ldots\} & \\
\text{CONST} & ::= & \{\texttt{3}, \texttt{false}, \ldots\} & \\
\text{PRIM} & ::= & \{\texttt{+}, \texttt{if}, \texttt{test-integer}, \ldots\} &
\end{array}$$

A program is a single lambda expression. The `letrec` form is used to define mutually recursive functions. Non-conditional primops like `+` and `cdr` take an extra continuation argument to call on their result: `(cdr x k)` calls procedure `k` on the cdr of `x`. Conditional branches are performed by special conditional primops which take multiple continuations. The `if` primop takes three arguments: `(if x c a)`. If the first argument `x` is a true value, the consequent continuation `c` is called; otherwise, the alternate continuation `a` is called. There is also a class of `test` primops that perform conditional type tests. For example, `(test-integer x c a)` branches to continuation `c` if `x` is an integer, otherwise to continuation `a`. Side effects on data structures are performed with appropriate primops, such as `set-car!`; side effects on variables are not allowed. CPS Scheme does not have the troublesome `call/cc` operator. When translating Scheme programs into their CPS Scheme representations, every occurrence of `call/cc` can be replaced with its CPS definition:

$$\texttt{(}\lambda\texttt{ (f k) (f (}\lambda\texttt{ (v k0) (k v)) k))}$$

Figure 2 shows a procedure that sums the first `n` integers in both standard Scheme and its CPS Scheme representation. It bears repeating that this extremely simple language is a practical and useful intermediate representation for full Scheme. In fact, the dialect presented here is essentially identical to the one used by the optimising Scheme compiler ORBIT.

For purposes of program analysis, let us extend this grammar by assuming that all expressions in a program are tagged with labels, drawn from some suitable set LAB:

$$\ell\texttt{:(}\lambda\texttt{ (x) }c\texttt{:(}r_1\texttt{:f }r_2\texttt{:x }k\texttt{:3 }r_3\texttt{:x))}$$

Each lambda, call, constant, and variable reference in this expression is tagged with a unique label. Expressions in a program that are identical receive distinct labels, so the two references to `x` have

```
(λ (n) (letrec ((lp (λ (i sum) (if (zero? i) sum
                                   (lp (- i 1) (+ i sum))))))
         (lp n 0)))

(λ (n k)
  (letrec ((lp (λ (i sum c)
                 (test-zero i
                   (λ () (c sum))
                   (λ ()
                     (+ sum i (λ (sum1)
                        (- i 1 (λ (i1)
                           (lp i1 sum1 c))))))))))
    (lp n 0 k)))
```

Figure 2: Standard and CPS Scheme to sum 1 through n

the different labels r_2 and r_3. Labels allow us to uniquely identify different pieces of a program. We will suppress them when convenient. A useful syntactic function is the *binder* function, which maps a variable to the label of its binding lambda or `letrec` construct, *e.g.*, *binder* $[\![\texttt{x}]\!] = \ell$.

3.2 NSAS

Casting our problem into CPS gives us a structure to work with; we now need a technique for analysing that structure. The method of non-standard abstract semantics (NSAS) is an elegant technique for formally describing program analyses. It forms the tool we'll need to solve our type recovery problem as described in the previous section. Section 8 gives several standard references for NSAS techniques.

Suppose we have a programming language L with a denotational semantics S, and we wish to determine some property X at compile time. Our first step is to develop an *alternate semantics* S_X for L that precisely expresses property X. That is, whereas semantics S might say the meaning of a program is a function "computing" the program's result value given its inputs, semantics S_X would say the meaning of a program is a function "computing" the property X on its corresponding inputs.

S_X is a precise definition of the property we wish to determine, but its precision typically implies that it cannot be computed at compile time. It might be uncomputable; it might also depend on the run-time inputs. The second step, then, is to *abstract* S_X to a new semantics, $\widehat{S}_X$ which trades accuracy for compile-time computability. This sort of approximation is a typical program-analysis tradeoff — the real answers we seek are uncomputable, so we settle for computable, conservative approximations to them.

The method of abstract semantic interpretation has several benefits. Since an NSAS-based analysis is expressed in terms of a formal semantics, it is possible to prove important properties about it. In particular, we can prove that the non-standard semantics S_X correctly expresses properties of the standard semantics S, and that the abstract semantics $\widehat{S}_X$ is computable, and safe with respect to S_X. Further, due to its formal nature, and because of its relation to the standard semantics of a programming language, simply expressing an analysis in terms of abstract semantic interpretations helps to clarify it.

The reader who is more comfortable with computer languages than denotational semantics equations should not despair. The equations presented in this paper can quite easily be regarded as interpreters in a functional disguise. The important point is that these “interpreters” do not compute a program’s actual value, but some other property of the program — in our case, the types of all the variable references in the program. We compute this with a non-standard, abstract “interpreter” that abstractly executes the program, collecting information about references as it goes.

4 Perfect Type Recovery

Following the NSAS approach, the first step towards type recovery is to define a "perfect" analysis that will capture the notion of type recovery. Our perfect semantics, which we will call PTREC, does not have to be computable; we will concern ourselves with a computable approximation in Section 5.

Perfect type recovery gives us a *type cache*:

> A *type cache* for a CPS Scheme program P is a function δ that, for each variable reference r and each context b over r, returns $\delta\langle r,\, b\rangle$, a type of all the values to which r could evaluate in context b.

(For now, we will be intentionally vague about what a "context" is; this will be made precise later.) Once we've computed a type cache, we can easily find the type for any variable reference $r{:}v$ in our program:

$$\text{RefType}[\![r{:}v]\!] = \bigsqcup_{b} \delta\langle r,\, b\rangle$$

4.1 Notation

D^* is used to indicate all vectors of finite length over the set D. Functions are updated with brackets: $e\,[a \mapsto b,\; c \mapsto d]$ is the function mapping a to b, c to d, and everywhere else identical to function e. Vectors are written $\langle a,\, p,\, z\rangle$. The ith element of vector v is written $v{\downarrow}i$. The power set of A is $\mathbf{P}(A)$. Function application is written with juxtaposition: $f\ x$. We extend a lattice's meet and join operations to functions into the lattice in a pointwise fashion, *e.g.*: $f \sqcap g = \lambda x.\ (f\ x) \sqcap (g\ x)$

4.2 Basic Domains

The PTREC semantics maps a CPS Scheme program to its type cache. Its structure, however, is very similar to a standard semantics for CPS Scheme. Let us first consider this basic structure without paying close attention to the parts of the semantics that actually track type information.

There is a basic domain, Bas, which consists of the integers and a special false value. (I will follow traditional Lisp practice in assuming no special boolean type; anything not false is a true value.) The domain of CPS Scheme procedures, Proc, has three parts: a primop is represented by its syntactic identifier (*prim* $\in$ PRIM), while a lambda closure is represented by a lambda/environment pair ($\langle \ell,\, \epsilon\rangle \in$ LAM $\times$ BEnv). The special token *stop* is the top-level continuation: in the standard semantics, calling *stop* on a value v halts the program with result v. The value domain D consists of the basic values and CPS Scheme procedures. The answer set TCache is the set of type caches.{Note No Bottom}

$$\begin{aligned}
\text{Bas} &= \mathcal{Z} + \{\text{false}\} \\
\text{Proc} &= (\text{LAM} \times \text{BEnv}) + \text{PRIM} + \{\textit{stop}\} \\
\text{D} &= \text{Proc} + \text{Bas} \\
\text{TCache} &= (\text{REF} \times \text{CN}) \rightarrow \text{Type}
\end{aligned}$$

Several items are conspicuously absent from these sets. This "toy" dialect omits i/o and a store, features that would be found in a full CPS Scheme semantics. It only provides three basic types of value: integers, false, and procedures. The run-time error checking has been left out of the semantics. These omissions are made to simplify the presentation. Extending the analysis to a more complete dialect of CPS Scheme is straightforward once the basic technique is understood. For example, the implementation described in Section 6 handles all of these missing features.

4.3 Environments and Procedures

The PTREC semantics factors the environment into two parts: the *variable environment* ($ve \in$ VEnv), which is a global structure, and the lexical *contour environment* ($\epsilon \in$ BEnv). A contour environment ϵ maps syntactic binding constructs — lambda and `letrec` expressions — to *contours* or dynamic frames. Each time a lambda is called, a new contour is allocated for its bound variables. Contours are taken from the set CN (the integers will suffice). A variable paired with a contour is a *variable binding* $\langle v,\ b\rangle$, taken from VB. The variable environment *ve*, in turn, maps these variable bindings to actual values. The contour part of the variable binding pair $\langle v,\ b\rangle$ is what allows multiple bindings of the same identifier to coexist peacefully in the single variable environment.

$$\begin{array}{rcl} \text{CN} & & \textit{Contours} \\ \text{VB} & = & \text{VAR} \times \text{CN} \\ \text{BEnv} & = & \text{LAB} \rightarrow \text{CN} \\ \text{VEnv} & = & \text{VB} \rightarrow \text{D} \end{array}$$

Lexical scoping semantics requires us to close a lambda expression with the contour environment that is present when the lambda is evaluated. We can see both the closure of lambda expressions and the lookup of variables in the $\mathcal{A}_v$ function below. $\mathcal{A}_v$ is the function that evaluates arguments in procedure calls, given the lexical contour environment ϵ and the global variable environment *ve*.

$$\mathcal{A}_v : \text{ARG} \cup \text{FUN} \rightarrow \text{BEnv} \rightarrow \text{VEnv} \rightarrow \text{D}$$

$$\begin{array}{rcl} \mathcal{A}_v\, [\![\texttt{(}\lambda\ \texttt{(}v_1 \ldots v_n\texttt{)}\ c\texttt{)}]\!]\, \epsilon\, ve & = & \langle [\![\texttt{(}\lambda\ \texttt{(}v_1 \ldots v_n\texttt{)}\ c\texttt{)}]\!],\ \epsilon \rangle \\ \mathcal{A}_v\, [\![v]\!]\, \epsilon\, ve & = & ve\, \langle v,\ \epsilon\, \mathit{binder}\, v \rangle \\ \mathcal{A}_v\, [\![\mathit{prim}]\!]\, \epsilon\, ve & = & \mathit{prim} \\ \mathcal{A}_v\, [\![k]\!]\, \epsilon\, ve & = & \mathcal{K}\, k \end{array}$$

$\mathcal{A}_v$ closes lambdas over the contour environment ϵ. Variable references are looked up in a two step process. First, the contour environment is indexed with the variable's binding lambda or `letrec` expression *binder v* to find which contour this reference occurs in. The contour and the variable are then used to index into the variable environment *ve*, giving the actual value. Since primops are denoted by their syntactic identifiers, $\mathcal{A}_v$ maps these to themselves. Constants are handled by some appropriate meaning function $\mathcal{K}$.

New contours are created when procedures are called. Procedure calls are handled by the $\mathcal{C}$ and $\mathcal{F}$ functions.

$$\mathcal{C}\ : \text{CALL} \rightarrow \text{BEnv} \rightarrow \text{VEnv} \rightarrow \text{QEnv} \rightarrow \text{TTab} \rightarrow \text{TCache}$$

$$\begin{aligned}
&\mathcal{C}\,[\![\texttt{(}e_0{:}f\ \ e_1{:}a_1 \ldots e_n{:}a_n\texttt{)}]\!]\ \epsilon\ ve\ qe\ \tau\ = \delta \sqcup\ (\mathcal{F}\, f')\ av\ qv\ ve\ qe\ \tau' \\
&\text{where } av{\downarrow}i = \ \mathcal{A}_v\ a_i\ \epsilon\ ve \\
&\qquad f' = \ \mathcal{A}_v\, f\ \ \epsilon\ ve \\
&\qquad \tau' = \begin{cases} \tau \sqcap \top\ [\,\mathcal{A}_q\, f\ \ \epsilon\ qe\ \mapsto \textit{type/proc}] & f\ \in \text{REF} \\ \tau & \textit{otherwise} \end{cases} \\
&\qquad \delta = [\langle e_i,\ \ \epsilon\ \textit{binder}\, e_i\ \rangle \mapsto \tau(\mathcal{A}_q\ e_i\ \epsilon\ qe\,)] \quad \forall [\![e_0{:}f\,]\!], [\![e_i{:}a_i]\!] \in \text{REF} \\
&\qquad qv{\downarrow}i = \begin{cases} \mathcal{A}_q\ a_i\ \epsilon\ qe & a_i \in \text{REF} \quad \text{(quantity)} \\ \mathcal{A}_t\ a_i & \textit{otherwise} \quad \text{(type)} \end{cases}
\end{aligned}$$

$\mathcal{C}$ takes five arguments: a call expression, the lexical contour environment ϵ, the variable environment *ve*, and two others used for type recovery (we will return to these last two arguments in the next subsection). $\mathcal{C}$ uses the $\mathcal{A}_v$ function to determine the values for the procedure being called f' and the arguments being passed to it. The argument values are collected into a single argument vector *av*. CPS Scheme procedures are represented by either lambda/environment pairs or by primop identifier names; the semantic function $\mathcal{F}$ converts this denotation of procedure f' to a functional value. The resulting function provides the contribution made to the final type cache by the execution of the program from the entry to procedure f' forward.

The secondary, functional representation of CPS Scheme procedures is produced by the $\mathcal{F}$ function:

$$\mathcal{F} : \text{Proc} \rightarrow \text{D}^* \rightarrow (\text{Quant} + \text{Type})^* \rightarrow \text{VEnv} \rightarrow \text{QEnv} \rightarrow \text{TTab} \rightarrow \text{TCache}$$

$$\begin{aligned}
&\mathcal{F}\ \langle [\![\ell{:}\texttt{(}\lambda\ \ \texttt{(}v_1 \ldots v_n\texttt{)}\ \ c\texttt{)}]\!], \epsilon\rangle\ = \\
&\quad \lambda av\ qv\ ve\ qe\ \tau.\ \ \mathcal{C}\ c\ \epsilon'\ ve'\ qe''\ \tau' \\
&\qquad\qquad \text{where}\ \ b = nb \\
&\qquad\qquad\qquad \epsilon' = \epsilon\ [\ell \mapsto b] \\
&\qquad\qquad\qquad ve' = ve\ [\langle v_i,\ b\rangle \mapsto av{\downarrow}i] \\
&\qquad\qquad\qquad qe' = qe\ [\langle v_i,\ b\rangle \mapsto qv{\downarrow}i] \quad \forall i\ \ni\ qv{\downarrow}i \in \text{Quant} \qquad (*) \\
&\qquad\qquad\qquad \left.\begin{aligned} qe'' &= qe'\ [\langle v_i,\ b\rangle \mapsto \langle v_i,\ b\rangle] \\ \tau' &= \tau\ [\langle v_i,\ b\rangle \mapsto qv{\downarrow}i] \end{aligned}\right\} \forall i\ \ni\ qv{\downarrow}i \in \text{Type} \quad (**)
\end{aligned}$$

A CPS Scheme lambda procedure is represented by a function that takes five arguments: an argument vector *av*, the variable environment *ve*, and three others concerned with type recovery. Upon entry to this function, a new binding contour *b* is allocated for the lambda's scope. The function *nb* is responsible for allocating the new binding contour; it is essentially a gensym, returning a unique value each time it is called {Note Gensym}. The lexical contour environment ϵ is updated to map the current procedure's label ℓ to this new contour. The mappings $[\langle v_i,\ b\rangle \mapsto av{\downarrow}i]$ are added to the variable environment, recording the binding of ℓ's parameters to the arguments passed in *av* {Note Run-time Errors}. We update the type-tracking values *qe* and τ, and call $\mathcal{C}$ to evaluate the lambda's internal call expression *c* in the new environment.

To fully specify $\mathcal{F}$, we must also give the functional representation for each primop and the terminal *stop* continuation. We will return to this after considering the mechanics of type-tracking. We also need to specify how $\mathcal{C}$ handles call forms that are `letrec` expressions instead of simple procedure calls. This case is simple: $\mathcal{C}$ just allocates a new contour *b* for its scope, closes the defined procedures in the new contour environment ϵ' (thus providing the necessary circularity), and evaluates the `letrec`'s interior call form c in the new environment {Note Non-circular `letrec`}.

$$
\begin{aligned}
\mathcal{C}\,[\![\ell\texttt{:(letrec ((}f_1\ l_1\texttt{)...)}\ c\texttt{)}]\!]\,\epsilon\ ve\ qe\ \tau \;=\; & \mathcal{C}\ c\ \epsilon'\ ve'\ qe'\ \tau' \\
& \text{where } b = nb \\
& \quad \epsilon' = \epsilon\,[\ell \mapsto b] \\
& \quad ve' = ve\,[\langle f_i, b\rangle \mapsto \mathcal{A}_v\ l_i\ \epsilon'\ ve\,] \\
& \quad qe' = qe\,[\langle f_i, b\rangle \mapsto \langle f_i, b\rangle] \\
& \quad \tau' = \tau\,[\langle f_i, b\rangle \mapsto \mathit{type/proc}]
\end{aligned}
$$

4.4 Quantities and Types

The semantics presented so far could easily be for a standard interpretation of CPS Scheme. We can now turn to the details of tracking type information through the PTREC interpretation. This will involve the quantity analysis model discussed in Section 2. The general type recovery strategy is straightforward:

- Whenever a new computational quantity is created, it is given a unique qname. Over the lifetime of a given quantity, it will be bound to a series of variables as it is passed around the program. As a quantity (from D) is bound to variables, we also bind its qname (from Quant) with these variables.
- As execution proceeds through the program, we keep track of all currently known information about quantities. This takes the form of a *type table* τ that maps qnames to type information. Program structure that determines type information about a quantity enters the information into the type table, passing it forward.
- When a variable reference is evaluated, we determine the qname it is bound to, and index into the type table to discover what is known at that point in the computation about the named quantity. This tells us what we know about the variable reference in the current context. This information is entered into the answer type cache.

This amounts to instrumenting our standard semantics to track the knowledge determined by run-time type tests, recording relevant snapshots of this knowledge in the answer type cache as execution proceeds through the program.

The first representational decision is how to choose qnames. A simple choice is to name a quantity by the first variable binding $\langle v, b\rangle$ to which it is bound. Thus, the qname for the cons cell created by

```
(cons 1 2 (λ (x) ...))
```

is $\langle[\![\texttt{x}]\!], b\rangle$, where b is the contour created upon entering `cons`'s continuation. When future variable bindings are made to this cons cell, the binding will be to the qname $\langle[\![\texttt{x}]\!], b\rangle$. Thus, our qname set Quant is just the set of variable bindings VB:

$$\text{Quant} = \text{VB}$$

Having chosen our qnames, the rest of the type-tracking machinery falls into place. A quantity environment ($qe \in$ QEnv) is a mapping from variable bindings to qnames. The qname analog of the variable environment *ve*, the quantity environment is a global structure that is augmented monotonically over the course of program execution. A type table ($\tau \in$ TTab) is a mapping from qnames to types. Our types are drawn from some standard type lattice; for this example, we use the obvious lattice over the three basic types: procedure, false, and integer.

$$\begin{aligned}
\text{Type} &= \{\textit{type/proc}, \textit{type/int}, \textit{type/false}, \bot, \top\} \\
\text{QEnv} &= \text{VB} \to \text{Quant} \\
\text{TTab} &= \text{Quant} \to \text{Type}
\end{aligned}$$

We may now consider the workings of the type-tracking machinery in the $\mathcal{F}$ and $\mathcal{C}$ functions. Looking at $\mathcal{F}$, we see that our function linkage requires three additional arguments to be passed to a procedure: the quantity vector *qv*, the quantity environment *qe*, and the current type table τ. The quantity environment and type table are as discussed above. The quantity vector gives quantity information about the arguments passed to the procedure. Each element of *qv* is either a qname or a type. If it is a qname, it names the quantity being passed to the procedure as its corresponding argument. However, if a computational quantity has been created by the procedure's caller, then it has yet to be named — quantities are named by the first variable binding to which they are bound, so it is the duty of the current procedure to assign a qname to the new quantity as it binds it. In this case, the corresponding element of the quantity vector gives the initial type information known about the new quantity. Consider the cons example given above. The `cons` primop creates a new quantity — a cons cell — and calls the continuation `(`λ` (x) ...)` on it. Since the cons cell is a brand new quantity, it has not yet been given a qname; the continuation binding it to `x` will name it. So instead of passing the continuation a qname for the cell, the `cons` primop passes the type *type/pair* in *qv*, giving the quantity's initial type information.

We can see this information being used in the $\mathcal{F}$ equation. The line marked with (*) binds qnames from the quantity vector to their new bindings $\langle v_i,\, b\rangle$. The lines marked with (**) handle new quantities which do not yet have qnames. A new quantity must be assigned its qname, which for the *i*th argument is just its variable binding $\langle v_i,\, b\rangle$. We record the initial type information ($qv{\downarrow}i \in$ Type) known about the new quantity in the type table τ'. The new quantity environment qe'' and type table τ' are passed forward to the $\mathcal{C}$ function.

$\mathcal{C}$ receives as type arguments the current quantity environment *qe*, and the current type table τ. Before jumping off to the called procedure, $\mathcal{C}$ must perform three type-related tasks:

- Record in the final answer cache the type of each variable reference in the call. Each variable reference e_i:a_i is evaluated to a qname by the auxiliary function $\mathcal{A}_q$, the qname analog to the $\mathcal{A}_v$ function. The qname is used to index into the type table τ, giving the type information currently known about the quantity bound to variable a_i. Call this type *t*. We record in the type cache that the variable reference e_i evaluated to a value of type *t* in context $\epsilon\,\textit{binder}\,e_i$. This is the contribution δ that $\mathcal{C}$ makes to the final answer for the current call.
- Compute the quantity vector *qv* to be passed to the called procedure *f* . If a_i is a variable reference, its corresponding element in *qv* is the qname it is bound to in the current context; this is computed by the $\mathcal{A}_q$ auxiliary. On the other hand, if the argument a_i is a constant or a lambda, then it is considered a new, as-yet-unnamed quantity. The auxiliary $\mathcal{A}_t$ determines its type; the called procedure will be responsible for assigning this value a qname.

- Finally, note that $\mathcal{C}$ can do a bit of type recovery. If f does not evaluate to a procedure, the computation becomes undefined at this call. We may thus assume that f's quantity is a procedure in following code. We record this information in the outgoing type table τ': if f is a variable reference, we find the qname it is bound to, and intersect *type/proc* with the type information recorded for the qname in τ. If f is not a variable reference, it isn't necessary to do this. (Note that in the `letrec` case, $\mathcal{C}$ performs similar type recovery, recording that the new quantities bound by the `letrec` all have type *type/proc*. This is all the type manipulation $\mathcal{C}$ does for the `letrec` case.)

$$\mathcal{A}_q\,[\![v]\!]\,\epsilon\, qe \quad = \quad qe\,\langle v,\ \epsilon\ binder\, v\ \rangle$$

$$\begin{aligned}
\mathcal{A}_t\,[\![\texttt{(}\lambda\ \texttt{(}v_1 \ldots v_n\texttt{)}\ c\texttt{)}]\!] &= \textit{type/proc}\\
\mathcal{A}_t\,[\![n]\!] &= \textit{type/int} \quad \text{(numeral } n\text{)}\\
\mathcal{A}_t\,[\![\texttt{false}]\!] &= \textit{type/false}
\end{aligned}$$

Most of the type information is recovered by the semantic functions for primops, retrieved by $\mathcal{F}$. As representative examples, I will show the definitions of **+** and `test-integer`.

$$\begin{aligned}
\mathcal{F}\,[\![\texttt{+}]\!] &= \lambda\langle a,\ b,\ c\rangle\ \langle qa,\ qb,\ qc\rangle\ ve\ qe\ \tau.\ \ (\mathcal{F}\ c)\ \langle a+b\rangle\ \langle ts\rangle\ ve\ qe\ \tau''\\
&\quad \text{where } ta = \text{ QT } qa\ \tau\\
&\quad\quad\quad tb = \text{ QT } qb\ \tau\\
&\quad\quad\quad ts = \text{infer}\texttt{+}\langle ta,\ tb\rangle\\
&\quad\quad\quad \tau' = \text{ Tu } qa\ (ta \sqcap \textit{type/int})\ \tau\\
&\quad\quad\quad \tau'' = \text{ Tu } qb\ (tb \sqcap \textit{type/int})\ \tau'
\end{aligned}$$

The **+** primop takes three arguments: two values to be added, a and b, and a continuation c. Hence the argument vector and quantity vector have three components each. The primop assigns ta and tb to be the types that execution has constrained a and b to have. These types are computed by the auxiliary function QT:(Quant + Type) $\rightarrow$(Quant $\rightarrow$ Type) $\rightarrow$ Type.

$$\text{QT}\ q\ \tau\ =\ \tau\ q \qquad \text{QT}\ t\ \tau\ = t$$

QT maps an element from a quantity vector to type information. If the element is a qname q, then QT looks up its associated type information in the type table τ. If the element is a type t (because the corresponding quantity is a new, unnamed one), then t is the initial type information for the quantity, and so QT simply returns t. Having retrieved the type information for its arguments a and b, **+** can then compute the type ts of its result sum. This is handled by the auxiliary function infer**+**, whose details are not presented. Infer**+** is a straightforward type computation: if both arguments are known to be integers, then the result is known to be an integer. If our language includes other types of numbers, infer**+** can do the related inferences. For example, it might infer that the result of adding a floating point number to another number is a floating point number. However infer**+** computes its answer, ts must be a subtype of the number type, since if control proceeds past the addition, **+** is guaranteed to produce a number. Further, **+** can make inferences about subsequent references to its arguments: they must be numbers (else the computation becomes undefined at the addition). The auxiliary function Tu updates the incoming type table with this inference; the result type table τ'' is passed forwards to the continuation.

$$\mathrm{Tu}\; q\; t\; \tau \;= \begin{cases} \tau & q \in \mathrm{Type} \\ \tau\,[q \to t] & q \in \mathrm{Quant} \end{cases}$$

Tu takes three arguments: an element q from a quantity vector (*i.e.*, a qname or type), a type t, and a type table τ. If q is a qname, the type table is updated to map $q \mapsto t$. Otherwise, the type table is returned unchanged (the corresponding quantity is ephemeral, being unnamed by a qname; there is no utility in recording type information about it, as no further code can reference it). With the aid of Tu, **+** constrains its arguments a and b to have the number type by intersecting the incoming type information *ta* and *tb* with *type/number* and updating the outgoing type table τ'' to reflect this{Note Recovering Continuations}. The sum $a + b$, its initial type information *ts*, and the new type table τ'' are passed forward to the continuation, thus continuing the computation. As mentioned earlier, we omit the case of halting the computation if there is an error in the argument values, *e.g.*, a or b are not numbers {Note Run-time Errors}.

$$\mathcal{F}\,[\![\texttt{test-integer}]\!] \;=\; \lambda\langle x,\, c,\, a\rangle\; \langle qx,\, qc,\, qa\rangle\; ve\; qe\; \tau.\; \begin{cases} (\mathcal{F}\; c)\; \langle\rangle\; \langle\rangle\; ve\; qe\; \tau_t & x \in \mathcal{Z} \\ (\mathcal{F}\; a)\; \langle\rangle\; \langle\rangle\; ve\; qe\; \tau_f & \textit{otherwise} \end{cases}$$

$$\begin{aligned} \text{where } tx &= \mathrm{QT}\; qx\; \tau \\ \tau_f &= \mathrm{Tu}\; qx\; (tx - \textit{type/int})\; \tau \\ \tau_t &= \mathrm{Tu}\; qx\; (tx \sqcap \textit{type/int})\; \tau \end{aligned}$$

The primop `test-integer` performs a conditional branch based on a type test. It takes as arguments some value x, and two continuations c and a. If x is an integer, control is passed to c, otherwise control is passed to a. `Test-integer` uses QT to look up *tx*, the type information recorded in the current type table τ for x's qname *qx*. There are two outgoing type tables computed, one which assumes the test succeeds (τ_t), and one which assumes the test fails (τ_f). If the test succeeds, then *qx*'s type table entry is updated by Tu to constrain it to have the integer type. Similarly, if the test fails, *qx* has the integer type subtracted from its known type. The appropriate type table is passed forwards to the continuation selected by `test-integer`, producing the answer type cache.

Finally, we come to the definition of the terminal *stop* continuation, retrieved by $\mathcal{F}$. Calling the *stop* continuation halts the program; no more variables are referenced. So the semantic function for *stop* just returns the bottom type cache $\perp$:

$$\mathcal{F}\,[\![\textit{stop}]\!] \;=\; \lambda\,\langle v\rangle\;\; qv\; ve\; qe\; \tau.\; \perp$$

The bottom type cache is the one that returns the bottom type for any reference: $\perp_{\mathrm{TCache}} = \lambda x.\; \perp_{\mathrm{Type}}$. Note that in most cases, the bottom type cache returned by calling *stop* is *not* the final type cache for the entire program execution. Each call executed in the course of the program execution will add its contribution to the final type cache. This contribution is the expression δ in the $\mathcal{C}$ equation for simple call expressions on page 11.

Having defined all the PTREC type tracking machinery, we can invoke it to compute the type cache δ for a program by simply closing the top-level lambda ℓ in the empty environment, and passing it the terminal *stop* continuation as its single argument:

$$\delta \;=\; (\mathcal{F}\;\langle \ell,\; \perp\rangle)\; \langle \textit{stop}\rangle\; \langle \textit{type/proc}\rangle\; \perp\; \perp\; \perp$$

5 Approximate Type Recovery

5.1 Initial Approximations

Having defined our perfect type recovery semantics, we can consider the problem of abstracting it to a computable approximation, while preserving some notion of correctness and as much precision as possible.

Conditional branches cannot, in general, be determined at compile time. So our abstract semantics must compute the type caches for both continuations of a conditional and union the results together. Note that this frees the semantics up from any dependence on the basic values Bas, since they are not actually tested by the semantics. Hence they can be dropped in the approximate semantics.

In addition, the infinite number of contours that a lambda can be closed over must be folded down to a finite set. The standard abstractions discussed in [CFASem] can be employed here: we can replace our variable environment with one that maps variable bindings to sets of procedures; and replace the contour allocation function *nb* with a function on lexical features of the program — *e.g.*, the contour allocated on entry to lambda $[\![\ell\texttt{:(}\lambda\ \texttt{(}v_1 \ldots v_n\texttt{)}\ c\texttt{)}]\!]$ can be the label ℓ of the lambda (what I call the "0th-order procedural approximation"). This allows multiple bindings of the same variable to be mapped together, allowing for a finite environment structure, which in turn gives a computable approximate semantics.

See [CFASem] for a detailed treatment of these abstractions. Both [CFASem] and [CFlow] discuss more precise alternatives to 0th-order approximation.

5.2 Problems with the Abstraction

It turns out that this standard set of approximations breaks the correctness of our semantics. The reason is that by folding together multiple bindings of the same variable, information can leak across quantity boundaries. For example, consider the following puzzle:

```
(let ((f (λ (g x) (if (integer? x) (g)
                      (λ () x)))))
  (f (f nil 3.7) 2))
```

Suppose that the procedural abstraction used by our analysis identifies together the contours created by the two calls to `f`. Consider the second execution of the `f` procedure: the variable `x` is tested to see if its value (2) is an integer. It is, so we jump to the value of `g`, which is simply `(λ () x)`. Now, we have established that `x` is bound to an integer, so we can record that this reference to `x` is an integer reference — which is an error, since `g = (λ () x)` is closed over a different contour, binding `x` to a non-integer, 3.7. We tested one binding of `x` and referred to a different binding of `x`. Our analysis got confused because we had identified these two bindings together, so that the information gathered at the test was erroneously applied at the reference.

This is a deep problem of the approximation. Quantity-based analyses depend upon keeping separate the information associated with different quantities; computable procedural approximations depend upon folding multiple environments together, confounding the separation required by a quantity-based analysis. I refer to this problem as the "environment problem" because it arises from our inability to precisely track environment information.

Only certain data-flow analyses are affected by the environment problem. The key property determining this is the direction in which the iterative analysis moves through the approximation lattice. In control-flow analysis, or useless-variable elimination [CFlow], the analysis starts with an overly-precise value and incrementally weakens it until it converges; all motion in the approximate lattice is in the direction of more approximate values. So, identifying two contours together simply causes further approximation, which is safe.

In the case of type recovery, however, our analysis moves in both directions along the type lattice as the analysis proceeds, and this is the source of the environment problem. When two different calls to a procedure, each passing arguments of different types, bind a variable in the same abstract contour, the types are joined together — moving through the type lattice in the direction of increasing approximation. However, passing through a conditional test or a primop application causes type information to be narrowed down – moving in the direction of increasing precision. Unfortunately, while it is legitimate to narrow down a variable's type in a single perfect contour, it is not legitimate to narrow down its type in the corresponding abstract contour — other bindings that are identified together in the same abstract contour are not constrained by the type test or primop application. This is the heart of the problem with the above puzzle.

In general, then, the simple abstraction techniques of [CFASem] yield correct, conservative, safe analyses only when the analysis moves through its answer lattice only in the direction of increasing approximation.

5.3 Control Flow Analysis

Before proceeding to a solution for the environment problem, we must define a necessary analysis tool, the *call context cache* provided by *control flow analysis*:

> A call context cache (*cc cache*) for a CPS Scheme program P is a function γ that, for every call site c in P and every environment ϵ over c gives $\gamma\ \langle c,\ \epsilon\rangle$, a conservative superset of the procedures called from c in environment ϵ during the execution of P.

This is a straightforward computation using the non-standard abstract semantic interpretation approach discussed above. Note that a cc cache is essentially a trace of program execution to some level of approximation — later we will exploit this property to "restart execution" at some arbitrary point in the computation. The cc cache is an approximation in two ways. First, the set it returns for a given call context is not required to be tight — it is only guaranteed to be a superset. Second, the environment structure ϵ that is the second component of a call context $\langle c,\ \epsilon\rangle$ and a closure $\langle l,\ \epsilon\rangle$ is a finite abstraction of the fully detailed environment structure.

The reader wishing a detailed account of control flow analysis should refer to [CFASem].

5.4 Perfect Contours

Our central problem is that we are identifying together different contours over the same variable. We are forced to this measure by our desire to reduce an infinite number of contours down to a finite, computable set. The central trick to solving this problem is to reduce the infinite set of contours down to a finite set, *one of which corresponds to a single contour in the perfect semantics.* Flow analysis then tracks this perfect contour, whose bindings will never be identified with any other contours over the same variable scope. Information associated with quantities bound in this perfect contour cannot be confounded. The other approximate contours are used only to provide the approximate control flow information for tracing through the program's execution. We still have only a finite number of contours — the finite number of approximate contours plus the one perfect contour — so our analysis is still computable.

For example, suppose we know that procedure $p = \langle [\![\ell\texttt{:(}\lambda\texttt{ (x y) ...)}]\!], \epsilon\rangle$ is called from call context $\langle [\![c\texttt{:(f 3 false)}]\!], \epsilon'\rangle$. We can do a partial type recovery for references to `x` and `y`. We perform a function call to *p*, creating a new perfect contour *b*. The variables we are tracking are bound in this contour, with variable bindings $\langle [\![\texttt{x}]\!], b\rangle$ and $\langle [\![\texttt{y}]\!], b\rangle$. We create new qnames for the arguments passed on this call to *p*, which are just the new perfect variable bindings $\langle [\![\texttt{x}]\!], b\rangle$ and $\langle [\![\texttt{y}]\!], b\rangle$. Our initial type table $\tau = [\langle [\![\texttt{x}]\!], b\rangle \mapsto \textit{type/int}, \langle [\![\texttt{y}]\!], b\rangle \mapsto \textit{type/false}]$ is constructed from what is known about the types of the arguments in *c* (this may be trivially known, if the arguments are constants, or taken from a previous iteration of this algorithm, if the arguments are variables).

We may now run our interpretation forwards, tracking the quantities bound in the perfect closure. Whenever we encounter a variable reference to `x` or `y`, if the reference occurs in the perfect contour *b*, then we can with certainty consult the current type table τ to obtain the type of the reference. Other contours over `x` and `y` won't confuse the analysis. Note that we are only tracking type information associated with the variables `x` and `y`, for a single call to ℓ. In order to completely type analyse the program, we must repeat our analysis for each lambda for each call to the lambda. This brings us to the Reflow semantics.

5.5 The Reflow Semantics

The abstract domains and functionalities of the Reflow semantics are given in Figure 3. The abstract domains are very similar to the perfect domains of the PTREC semantics, and show the approximations discussed in Subsection 5.1: basic values have been dropped, the contour set $\widehat{\mathrm{CN}}$ is finite, and elements of $\widehat{\mathrm{D}}$ are sets of abstract procedures, not single values.

The idea of the Reflow semantics is to track only one closure's variables at a time. This is done by the *Reflow* function:

$$\textit{Reflow} : \widehat{\mathrm{CC}} \rightarrow \widehat{\mathrm{VEnv}} \rightarrow \mathrm{Type}^* \rightarrow \mathrm{T}\widehat{\mathrm{Cache}}$$

For example, $\textit{Reflow}\langle [\![\texttt{(f a b)}]\!], \epsilon\rangle \textit{ve}\langle \textit{type/int}, \textit{type/proc}\rangle$ restarts the program at the call `(f a b)`, in the context given by the (approximate) contour environment ϵ and variable environment *ve*, assuming `a` has type *type/int* and `b` has type *type/proc*. *Reflow* runs the program forward, tracking only the variables bound by the initial call to `f`, returning a type cache giving information about references to these variables. This is done by allocating a single, perfect contour for the initial

$$\begin{aligned}
\widehat{\text{Proc}} &= (\text{LAM} \times \widehat{\text{BEnv}}) + \text{PRIM} + \{\mathit{stop}\}\\
\widehat{\text{D}} &= \mathbf{P}(\widehat{\text{Proc}})\\
\widehat{\text{TCache}} &= (\text{REF} \times \widehat{\text{CN}}) \rightarrow \text{Type}\\
\widehat{\text{CC}} &= \text{CALL} \times \widehat{\text{BEnv}}\\
\widehat{\text{CN}} &= \text{LAB} \cup \{\langle \mathit{perfect},\ l\rangle \mid l \in \text{LAB}\}\\
\widehat{\text{VB}} &= \text{VAR} \times \widehat{\text{CN}}\\
\widehat{\text{BEnv}} &= \text{LAB} \rightarrow \widehat{\text{CN}}\\
\widehat{\text{VEnv}} &= \widehat{\text{VB}} \rightarrow \widehat{\text{D}}\\
\\
\widehat{\text{Quant}} &= \widehat{\text{VB}}\\
\widehat{\text{TTab}} &= \widehat{\text{Quant}} \rightarrow \text{Type}
\end{aligned}$$

$$\begin{aligned}
\widehat{\mathcal{C}} :&\quad \text{CALL} \rightarrow \widehat{\text{BEnv}} \rightarrow \widehat{\text{VEnv}} \rightarrow \widehat{\text{TTab}} \rightarrow \widehat{\text{TCache}}\\
\widehat{\mathcal{F}} :&\quad \widehat{\text{Proc}} \rightarrow \widehat{\text{D}}^{*} \rightarrow (\widehat{\text{Quant}}_{\perp})^{*} \rightarrow \widehat{\text{VEnv}} \rightarrow \widehat{\text{TTab}} \rightarrow \widehat{\text{TCache}}\\
\widehat{\mathcal{F}_p} :&\quad \widehat{\text{Proc}} \rightarrow \widehat{\text{D}}^{*} \rightarrow \text{Type}^{*} \rightarrow \widehat{\text{VEnv}} \rightarrow \widehat{\text{TCache}}
\end{aligned}$$

Figure 3: Abstract Domains and Functionalities

procedure called from `(f a b)`, and tracking the variables bound in this contour through an abstract execution of the program.

The initial type vector given to *Reflow* comes from an auxiliary function, *TVInit*:

$$\begin{aligned}
&\mathit{TVInit} : \widehat{\text{CC}} \rightarrow \widehat{\text{TCache}} \rightarrow \text{Type}^{*}\\
&\mathit{TVInit}\ \langle [\![\texttt{(}f\ \ a_1 \ldots a_n\texttt{)}]\!],\ \epsilon\rangle\ \delta\ = \langle t_1 \ldots t_n\rangle\\
&\qquad\qquad \text{where } t_i = \begin{cases} \delta\ \langle a_i,\ \epsilon\, \mathit{binder}\, a_i\rangle & a_i \in \text{REF}\\ \mathcal{A}_t\ a_i & a_i \notin \text{REF}\end{cases}
\end{aligned}$$

TVInit takes a call context and a type cache, and returns the types of all the arguments in the call. If the argument is a variable reference, the type cache is consulted; if the argument is a constant, lambda, or primop, the auxiliary function $\mathcal{A}_t$ gives the appropriate type.

If we wish to restart our program at an arbitrary call context $\langle c,\ \epsilon\rangle$ with *Reflow*, we require the variable environment *ve* that pertained at this point of call. This is easy to handle: we always use the final variable environment that was present at the end of the control-flow analysis of Subsection 5.3. Since the variable environment is only augmented monotonically during the course of executing a program, the terminal environment is a superset of all intermediate variable environments. So, our initial control-flow analysis computes two items critical for the Reflow analysis: the call cache γ and terminal variable environment $\mathit{ve}_{\mathit{final}}$.

Given *TVInit* and *Reflow*, we can construct a series of approximate type caches, converging to a fixed point. The initial type cache δ_0 is the most precise; at each iteration, we redo the reflow analysis assuming type cache δ_i, computing a weaker type cache δ_{i+1}. The limit δ is the final result.

$$\widehat{\mathcal{A}_v}\,[\![\ell\texttt{:(}\lambda\ \texttt{(}v_1 \ldots v_n\texttt{)}\ \ c\texttt{)}]\!]\,\epsilon\, ve \;=\; \{\langle \ell,\, \epsilon\rangle\} \qquad \widehat{\mathcal{A}_v}\,[\![prim]\!]\,\epsilon\, ve \;=\; \{prim\}$$

$$\widehat{\mathcal{A}_v}\,[\![v]\!]\,\epsilon\, ve \;=\; ve\,\langle v,\, \epsilon\, binder\, v\rangle \qquad \widehat{\mathcal{A}_v}\,[\![k]\!]\,\epsilon\, ve \;=\; \emptyset$$

$$\widehat{\mathcal{A}_q}\,[\![v]\!]\,\epsilon \;=\; \begin{cases} \langle v,\, b\rangle & perfect?\, b \\ \bot & otherwise \end{cases} \qquad (b = \epsilon\, binder\, v)$$

$$ptoa\,\langle perfect,\, b\rangle \;=\; b$$

$$perfect?\,\langle perfect,\, b\rangle \;=\; true \qquad perfect?\, b \;=\; false$$

Figure 4: Abstract Auxiliary Functions

The recomputation of each successive type cache is straightforward: for every call context $\langle c,\, \epsilon\rangle$ in the domain of call cache γ (that is, every call context recorded by the control-flow analysis of Subsection 5.3), we use the old type cache to compute the types of the arguments in the call, then reflow from the call, tracking the variables bound by the call's procedure, assuming the new type information. The returned type caches are joined together, yielding the new type cache. So the new type cache is the one we get by assuming the type assignments of the old type cache. A fixed point is a legitimate type assignment. Since all of our abstract domains are of finite size, and our type lattice has finite height, the least fixed point is computable.

$$\begin{aligned} \delta_0 &= \lambda\,\langle r,\, b\rangle\,.\,\bot \\ \delta_{i+1} &= \delta_i \;\sqcup \bigsqcup_{\langle c,\,\epsilon\rangle \in\, Dom\,\gamma} Reflow\,\langle c,\, \epsilon\rangle\, ve_{final}\,(\,TVInit\,\langle c,\, \epsilon\rangle\, \delta_i\,) \\ \delta &= \bigsqcup_i \delta_i \end{aligned}$$

Before we get to the machinery of the *Reflow* function itself, let us define a few useful auxiliary functions and concepts (Figure 4). Because the Reflow semantics has a single perfect contour coexisting with the approximate contours, we need a few utility functions for manipulating the two different kinds of contour. The approximate contours are simply the labels of all the syntactic binding constructs (lambdas and `letrec`'s): in the 0th-order approximation, the contour allocated when entering lambda $[\![\ell\texttt{:(}\lambda\ \texttt{(}v_1 \ldots v_n\texttt{)}\ \ c\texttt{)}]\!]$ is just ℓ, so all contours over a single lambda are identified together. For every approximate contour l, we want to have a corresponding perfect contour $\langle perfect,\, l\rangle$. These perfect contours are pairs marked with the token *perfect*. The predicate *perfect?* distinguishes perfect contours from approximate ones. The function *ptoa* strips off the perfect token, mapping a perfect contour to its approximate counterpart. The $\widehat{\mathcal{A}_v}$ function evaluates call arguments, and is the straightforward abstraction of its counterpart in the PTREC semantics. The $\widehat{\mathcal{A}_q}$ function is a little more subtle. Since we only track variables bound in perfect contours in the Reflow semantics, $\widehat{\mathcal{A}_q}$ only returns quantities for these bindings; approximate bindings are mapped to an undefined value, represented with $\bot$.

Now we are in a position to examine the machinery that triggers off a single wave of perfect contour type tracking: the *Reflow* function, and its auxiliary $\widehat{\mathcal{F}_p}$ function.

$$\begin{aligned}
\mathit{Reflow}\ \langle [\![c{:}\texttt{(}f\ \ a_1 \ldots a_n\texttt{)}]\!],\ \epsilon\rangle\ \mathit{ve}\ \mathit{tv} \quad &= \quad \bigsqcup_{f' \in F} (\widehat{\mathcal{F}_p} f')\ \mathit{av}\ \mathit{tv}\ \mathit{ve} \\
&\quad\ \ \text{where } F = \widehat{\mathcal{A}_v}\ f\ \ \epsilon\ \mathit{ve} \\
&\quad\ \ \qquad\ \ \mathit{av}{\downarrow}i = \widehat{\mathcal{A}_v}\ a_i\ \epsilon\ \mathit{ve}
\end{aligned}$$

Reflow simply reruns the interpretation from each possible procedure that could be called from call context $\langle c,\ \epsilon\rangle$. Each procedure is functionalised with the "perfect" functionaliser $\widehat{\mathcal{F}_p}$, who arranges for the call to the procedure to be a perfect one. The type caches resulting from each call are joined together into the result cache.

$$\begin{aligned}
\widehat{\mathcal{F}_p}\ \langle [\![\ell{:}\texttt{(}\lambda\ \ \texttt{(}v_1 \ldots v_n\texttt{)}\ \ c\texttt{)}]\!],\ \epsilon\rangle\ &= \lambda \mathit{av}\ \mathit{tv}\ \mathit{ve}.\ \widehat{\mathcal{C}}\ c\ \epsilon'\ \mathit{ve}'\ \tau \\
&\quad \text{where } b = \langle \mathit{perfect},\ \ell\rangle \\
&\qquad\quad \tau = [\langle v_i,\ b\rangle \mapsto \mathit{tv}{\downarrow}i] \\
&\qquad\quad \epsilon' = \epsilon\,[\ell \mapsto b] \\
&\qquad\quad \mathit{ve}' = \mathit{ve} \sqcup [\langle v_i,\ b\rangle \mapsto \mathit{av}{\downarrow}i]
\end{aligned}$$

The perfect functionalisation of a procedure produced by $\widehat{\mathcal{F}_p}$ is called only once, at the beginning of the reflow. A new contour is allocated, whose value is marked with the special *perfect* token to designate it the one and only perfect contour in a given execution thread. The incoming values are bound in the outgoing variable environment ve' under the perfect contour. The incoming type information is used to create the initial type table τ passed forwards to track the values bound under the perfect contour. The rest of the computation is handed off to the $\widehat{\mathcal{C}}$ procedure. $\widehat{\mathcal{C}}$ is similar to $\mathcal{C}$ with the exception that it only records the type information of references that are bound in the perfect contour.

$\widehat{\mathcal{F}_p}$ must also be defined over primops. Primops do not have variables to be type-recovered, so instead primops pass the buck to their continuations. The `+` primop uses its initial-type vector $\langle \mathit{ta},\ \mathit{tb},\ \mathit{tc}\rangle$ to compute the initial-type vector $\langle \mathit{ts}\rangle$ for its continuation. The `+` primop then employs $\widehat{\mathcal{F}_p}$ to perform type recovery on the variable bound by its continuation. The `test-integer` primop is even simpler. Since its continuations do not bind variables, there is nothing to track, so the function just immediately returns the bottom type cache.

$$\begin{aligned}
\widehat{\mathcal{F}_p}\ [\![\texttt{+}]\!] \quad &= \quad \lambda\,\langle a,\ b,\ c\rangle\ \ \langle \mathit{ta},\ \mathit{tb},\ \mathit{tc}\rangle\ \ \mathit{ve}.\ \bigsqcup_{c' \in c} (\widehat{\mathcal{F}_p} c')\ \langle \emptyset\rangle\ \langle \mathit{ts}\rangle\ \mathit{ve} \\
&\quad\ \ \text{where } \mathit{ts} = \text{infer}\texttt{+}\ \langle \mathit{ta},\ \mathit{tb}\rangle
\end{aligned}$$

$$\widehat{\mathcal{F}_p}\ [\![\texttt{test-integer}]\!] = \lambda\,\langle x,\ c,\ a\rangle\ \ \langle \mathit{tx},\ \mathit{tc},\ \mathit{ta}\rangle\ \ \mathit{ve}.\ \bot$$

Once the initial call to $\widehat{\mathcal{F}_p}$ has triggered a wave of type recovery for a particular lambda's variables, the actual tracking of type information through the rest of the program execution is handled by the $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{C}}$ functions. Most of the action happens in the $\widehat{\mathcal{C}}$ function.

$$\widehat{\mathcal{C}}\,[\![\texttt{(}e_0{:}f\ \ e_1{:}a_1 \ldots e_n{:}a_n\texttt{)}]\!]\,\epsilon\, ve\, \tau \;=\; \delta \sqcup \left(\bigsqcup_{f' \in F} (\widehat{\mathcal{F}} f')\; av\; qv\; ve\; \tau' \right)$$

$$\begin{aligned}
\text{where }\; F &= \widehat{\mathcal{A}_v}\, f\;\, \epsilon\; ve \\
b_i &= \epsilon\; binder\, a_i \quad (\forall a_i \in \text{REF}) \\
b_0 &= \epsilon\; binder\, f \quad (\text{if } f\, \in \text{REF}) \\
av{\downarrow} i &= \widehat{\mathcal{A}_v}\; a_i\; \epsilon\; ve \\
qv{\downarrow} i &= \widehat{\mathcal{A}_q}\; a_i\; \epsilon \\
\delta &= \left[\langle e_i,\; ptoa\; b_i \,\rangle \mapsto \tau\, \langle a_i,\, b_i \rangle \right] \quad (\forall b_i \ni\; perfect?\, b_i\,) \\
\tau' &= \begin{cases} \tau \sqcap \left[\langle f\,,\, b_0 \rangle \mapsto type/proc \right] & perfect?\, b_0 \\ \tau & otherwise \end{cases}
\end{aligned}$$

As $\widehat{\mathcal{C}}$ evaluates its arguments, it checks to see if any are variables whose types are being tracked. A variable a_i is being tracked if it is closed in a perfect contour, that is, if *perfect?* b_i is true, where $b_i = \epsilon\; binder\, a_i$. If an argument is being tracked, we look up its current type $\tau\,\langle a_i,\, b_i\rangle$, and record this in $\widehat{\mathcal{C}}$'s contribution δ to the type cache (recording the reference under the perfect contour b_i's abstraction *ptoa* b_i). The rest of $\widehat{\mathcal{C}}$'s stucture is similar to the perfect variant of Section 4. An outgoing type table τ' is constructed, reflecting that f must be of type *type/proc* (Again, note that this fact is only recorded in τ' if f is a variable currently being tracked).

Note also that since multiple contours are identified together in the abstract semantics, values in the approximate domain are *sets* of abstract procedures. Because of this, the call must branch to each of the possible procedures f' the function expression f could evaluate to. The result type caches are then all joined together.

The function returned by $\widehat{\mathcal{F}}$ constructs approximate contours when called. Because multiple environments are identified together by $\widehat{\mathcal{F}}$'s functionalised value, it cannot track type information for the variables bound by its procedure. Hence $\widehat{\mathcal{F}}$ has a fairly simple definition when applied to a closure, just augmenting the environment structure and passing the closure's body c off to $\widehat{\mathcal{C}}$. Note that the environment is updated by unioning a parameter's value set $av{\downarrow}i$ to the set already bound under the abstract contour.

$$\widehat{\mathcal{F}}\,\langle [\![\ell{:}\texttt{(}\lambda\ \ \texttt{(}v_1 \ldots v_n\texttt{)}\ \ c\texttt{)}]\!],\; \epsilon\rangle \;=\; \lambda av\; qv\; ve\; \tau.\;\; \widehat{\mathcal{C}}\; c\; \epsilon'\; ve'\; \tau$$

$$\begin{aligned}
\text{where }\; b &= l \quad \text{(0th order proc. approx.)} \\
\epsilon' &= \epsilon\,\left[\ell \mapsto b\right] \\
ve' &= ve \sqcup \left[\langle v_i,\; b\rangle \mapsto av{\downarrow} i \right]
\end{aligned}$$

$\widehat{\mathcal{F}}$'s definition for the terminal *stop* continuation is, again, trivial, ignoring its argument v and returning the bottom type cache:

$$\widehat{\mathcal{F}}\; [\![stop]\!] \;=\; \lambda\,\langle v\rangle\;\; qv\; ve\; \tau.\; \bot$$

$\widehat{\mathcal{F}}$'s behavior on primops is more interesting. If the argument x being passed to `test-integer` is being tracked (*i.e.*, qx is a quantity, not bottom), then we intersect *type/int* with qx's incoming type, passing the result type table τ_t to the true continuation, and we subtract *type/int* from qx's type in τ_f the table passed to the false continuation. In other words, we do the type recovery of the PTREC semantics, but only for the values being tracked.

$$\widehat{\mathcal{F}}[\![\texttt{test-integer}]\!] = \lambda\,\langle x, c, a\rangle\ \langle qx, qc, qa\rangle\ ve\,\tau.\ \left(\bigsqcup_{c'\in c} (\widehat{\mathcal{F}}c')\,\langle\rangle\,\langle\rangle\ ve\,\tau_t\right) \sqcup \left(\bigsqcup_{a'\in a} (\widehat{\mathcal{F}}a')\,\langle\rangle\,\langle\rangle\ ve\,\tau_f\right)$$

$$\text{where } \tau_f = \begin{cases} \tau\,[qx \mapsto (\tau\,qx - \textit{type/int})] & qx \neq \bot \\ \tau & \textit{otherwise} \end{cases}$$

$$\tau_t = \begin{cases} \tau\,[qx \mapsto (\tau\,qx \sqcap \textit{type/int})] & qx \neq \bot \\ \tau & \textit{otherwise} \end{cases}$$

The `+` primop is similar. If the arguments a and b are being tracked, then we update the type table passed forwards, otherwise we simply pass along the incoming type table τ unchanged. Since the continuations c' are functionalised with the approximate functionaliser, $\widehat{\mathcal{F}}$, the quantity vector is $\langle\bot\rangle$ — we will not be tracking the variables bound by the call to c'.

$$\widehat{\mathcal{F}}\,[\![\texttt{+}]\!] \quad = \quad \lambda\,\langle a, b, c\rangle\ \langle qa, qb, qc\rangle\ ve\,\tau.\ \bigsqcup_{c'\in c} (\widehat{\mathcal{F}}c')\,\langle\emptyset\rangle\,\langle\bot\rangle\ ve\,\tau''$$

$$\text{where } \tau' = \begin{cases} \tau\,[qa \mapsto (\textit{type/number} \sqcap \tau\,qa)] & qa \neq \bot \\ \tau & \textit{otherwise} \end{cases}$$

$$\tau'' = \begin{cases} \tau'\,[qb \mapsto (\textit{type/number} \sqcap \tau'\,qb)] & qb \neq \bot \\ \tau' & \textit{otherwise} \end{cases}$$

To finish off the Reflow semantics, we must take care of `letrec`. Abstracting $\mathcal{C}$'s definition for `letrec` is simple. Evaluating the `letrec`'s bound expressions only involves closing lambdas, not referencing variables. So the `letrec` will not "touch" any of the variables we are currently tracking. Hence the `letrec` does not make any local contribution to the answer type cache, but simply augments the variable environment *ve* with the procedure bindings and recursively evaluates the inner call *c*.

$$\widehat{\mathcal{C}}\,[\![\ell\texttt{:(letrec ((}f_1\ l_1\texttt{)...)}\ c\texttt{)}]\!]\,\epsilon\,ve\,\tau \ = \ \widehat{\mathcal{C}}\,c\,\epsilon'\,ve'\,\tau$$
$$\text{where } b = \ell$$
$$\epsilon' = \epsilon\,[\ell \mapsto b]$$
$$ve' = ve \sqcup \left[\langle f_i, b\rangle \mapsto \widehat{\mathcal{A}_v}\,l_i\,\epsilon'\,ve\right]$$

One detail of `letrec` that we have neglected is tracking the types of the variables bound by `letrec`. There are several ways to handle this. We could add a case to the *Reflow* function to handle reflowing from `letrec` expressions, creating a perfect contour for the `letrec`'s binding. This is a fairly complex and expensive way to handle a simple case. Because `letrec` is syntactically restricted to binding only lambda procedures to its variables, we can statically analyse this case, and simply assign in advance the procedure type to all references to all `letrec` variables. The simplest place to insert this static assignment is in the initial type cache δ_0 used in the *Reflow* iteration:

$$\delta_0 = \lambda\,\langle[\![r{:}v]\!], b\rangle\,.\ \begin{cases} \textit{type/proc} & \textit{binder}\,v \in \text{CALL} & (\texttt{letrec}) \\ \bot & \textit{binder}\,v \in \text{LAM} & (\text{lambda}) \end{cases}$$

This performs the type analysis of the `letrec` variables in one step, leaving the rest of the Reflow semantics free to concentrate on the variables bound by lambdas.

6 Implementation

I have a prototype implementation of the type recovery algorithm written in Scheme. It analyzes Scheme programs that have been converted into CPS Scheme by the front end of the ORBIT compiler [ORBIT]. The type-recovery code is about 900 lines of heavily commented Scheme; the control-flow analysis code is about 450 lines.

The implemented semantics features a store (allowing side-effects) and a type lattice that includes the symbol, pair, false, procedure, fixnum, bignum, flonum, vector, list, integer, and number types. Procedures are approximated using the first-order procedural abstraction (1CFA) of [CFASem]. In addition, CPS-level continuations are syntactically marked by the front-end CPS converter; this information is used to partition the procedure domain. This partition appears to greatly reduce the size of the sets propagated through the analysis, improving both the speed and precision of the analysis.

The implementation is for the most part a straightforward transcription of the approximate type recovery semantics. The variable environment, store, quantity environment, and result type cache are all kept as global data structures that are monotonically augmented as the analysis progresses.

The recursive semantic equations are realised as a terminating Scheme program by memoising the recursive $\widehat{\mathcal{C}}$ applications; when the Scheme $\widehat{\mathcal{C}}$ procedure is applied to a memoised set of arguments, it returns without making further contributions to the answer type cache. This is the "memoised pending analysis" technique discussed in [Fixpoints].

Little effort has been made overall to optimize the implementation. Still, the current analyzer runs acceptably well for small test cases; response time has been sufficiently quick in the T interpreter that I have not felt the need to compile it. At this point in my experimentation, there is no reason to believe that efficiency of the analysis will be an overriding issue in practical application of the type recovery algorithm.

The allure of type recovery, of course, is type-safe Scheme implementations with little run-time overhead. It remains to be seen whether there is enough recoverable type information in typical code to allow extensive optimisation. The algorithm has not been tested extensively on a large body of real code. However, early results are encouraging. As an example of the algorithm's power, it is able to completely recover the types of all variable references in the `delq` and `fact` procedures given in Section 1.

7 Discussion and Speculation

This section is a collection of small discussions and speculations on various aspects of Scheme type recovery.

7.1 Side Effects and External Procedures

The PTREC and Reflow semantics in this paper are toy semantics in that side-effects and external procedures have been explicitly left out to simplify the already excessively unwieldy equations. Restoring them is not difficult. We can adopt a simple model of side-effects where all procedural values placed into the store can be retrieved by any operator that accesses the store. These procedures are called "escaped procedures." We can also introduce the idea of unknown external procedures by introducing a special "external procedure" and a special "external call." Any value passed to the external procedure escapes; all escaped procedures can be called from the external call.

This model of side-effects and external procedures is discussed in detail in [CFlow]. More precise models, of course, are possible [RefCount].

The implementation discussed in Section 6 uses this simple model of side-effects and external procedures. The store is represented as a single set of escaped procedures. Because the store is only monotonically augmented during the course of the analysis, it is represented as a global variable that is an implicit argument to the $\widehat{\mathcal{C}}$ function. Because of the monotonic property of the store, the memoised pending analysis actually memoises a last-modified timestamp for the store, which greatly increases the efficiency of the memoising. This trick is also used for the global variable environment *ve*.

7.2 Safe and Unsafe Primops

A given implementation of Scheme chooses whether to provide "safe" primops, which are defined to cause a graceful error halt when applied to illegal values, or "dangerous" primops, which simply cause undefined effects when applied to illegal values. For example, most Scheme compilers efficiently open-code car as a single machine operation. Without compile-time type recovery to guarantee the type of the argument to a car application, this fast implementation is dangerous.

Type recovery can accept either safe or dangerous primops, or a combination of both{Note Recovering Primops}. In both cases, the primop semantics allows flow-analysis based type-recovery. However, while it is possible to recover types given dangerous primops, the analysis is of limited value. The information provided by type recovery has two basic uses:

- Eliminating run-time error checks from safe primop applications.
- Specialising generic primops based on their argument types (*e.g.*, converting a generic arithmetic operation to an integer operation).

In a dangerous implementation, the first of these uses does not apply. As we shall see below, specialising generic arithmetic is of limited utility as well.

7.3 Limits of the Type System

From the optimising compiler's point of view, the biggest piece of bad news in the Scheme type system is the presence of arbitrary-precision integers, or "bignums." Scheme's bignums, an elegant convenience from the programmer's perspective, radically interfere with the ability of type recovery to assign small integer "fixnum" types to variable references. The unfortunate fact is that two's complement fixnums are not closed under any of the common arithmetic operations. Clearly adding, subtracting, and multiplying two fixnums can overflow into a bignum. Less obvious is that simple negation can overflow: the most negative fixnum overflows when negated. Because of this, not even fixnum division is safe: dividing the most negative fixnum by -1 negates it, causing overflow into bignums. Thus, the basic fixnum arithmetic operations cannot be safely implemented with their corresponding simple machine operations. This means that most integer quantities cannot be inferred to be fixnums. So, even though type recovery can guarantee that all the generic arithmetic operations in Figure 1's factorial function are integer operations, this does not buy us a great deal.

Not being able to efficiently implement safe arithmetic operations on fixnums is terrible news for loops, because many loops iterate over integers, particularly array-processing loops. Taking five instructions just to increment a loop counter can drastically affect the execution time of a tight inner loop.

There are a few approaches to this problem:

- **Range analysis**
  Range analysis is a data-flow analysis technique that bounds the values of numeric variables [Range]. For example, range analysis can tell us that in the body of the following C loop, the value of `i` must always lie in the range $[0, 10)$:

  ```
  for(i=9; i>=0; i--) printf("%d ", a[i]);
  ```

  Range analysis can probably be applied to most integer loop counters. Consider the `strindex` procedure below:

  ```
  (define (strindex c str)
    (let ((len (string-length str)))
      (letrec ((lp (lambda (i)
                     (cond ((>= i len) -1) ; lose
                           ((char= c (string-ref str i)) i) ; win
                           (else (loop (+ i 1))))))) ; loop
        (lp 0))))
  ```

  Type recovery can guarantee that `len`, being the result of the `string-length` primop, is a fixnum. Range analysis can show that `i` is bounded by 0 and a fixnum; this is enough information to guarantee that `i` is a fixnum. Range analysis is useful in its own right as well — in this example, it allows us to safely open code the character access `(string-ref str i)` with no bounds check.

- **Abstract Safe Useage Patterns**
  The poor man's alternative to range analysis is to take the useage patterns that are guaranteed to be fixnum specific, and package them up for the user as syntactic or procedural abstractions.

These abstractions can be carefully decorated with `proclaim` declarations to force fixnum arithmetic. For example, a loop macro which has a `(for c in-string str)` clause can safely declare the string's index variable as a fixnum. This approach can certainly pick up string and array index variables.

- **Disable Bignums**
  Another cheap alternative to range analysis is to live dangerously and provide a compiler switch or declaration which allows the compiler to forget about bignums and assume all integers are fixnums. Throwing out bignums allows simple type recovery to proceed famously, and programs can be successfully optimised (successfully, that is, until some hapless user's program overflows a fixnum...).

- **Hardware support**
  Special tag-checking hardware, such as is provided on the SPARC, Spur and Lisp Machine architectures [SPARC][Spur][LispM], or fine-grained parallelism, such as is provided by VLIW architectures [Bulldog] [Fisher], allow fixnum arithmetic to be performed in parallel with the bignum/fixnum tag checking. In this case, the limitations of simple type recovery are ameliorated by hardware assistance. {Note VLIW}

7.4 Declarations

The dangerous `proclaim` declaration is problematic. A purist who wants to provide a guaranteed safe Scheme implementation might wish to ban `proclaim` on the grounds that it allows the user to write dangerous code. A multithreaded, single address-space PC implementation of Scheme, for example, might rely on run-time safety to prevent threads from damaging each other's data. Including `proclaim` would allow the compilation of code that could silently trash the system, or access and modify another thread's data.

On the other hand, safe declarations like `enforce` have limits. Some useful datatypes cannot be checked at run time. For example, while it is possible to test at run time if a datum is a procedure, it is *not* possible, in general, to test at run time if a datum is a procedure that maps floating-point numbers to floating-point numbers. Allowing the user to make such a declaration can speed up some critical inner loops. Consider the floating-point numeric integrator below:

```
(define (integ f x0 x1 n)
  (enforce fixnum? n)  (enforce procedure? f)
  (enforce flonum? x0) (enforce flonum? x1)
  (let ((delta (/ (- x1 x0) n)))
    (do ((i n (- i 1))
         (x x0 (+ x delta))
         (sum 0.0 (+ sum (f x))))
        ((= i 0) (* sum delta)))))
```

In some cases, analysis might be able to find all applications of the integrator, and thus discover that `f` is always bound to a floating-point function. However, if the integrator is a top level procedure in an open system, we can't guarantee at compile time that `f` is a floating-point function, and we can't enforce it at run time. This means that the sum operation must check the return value of `f` each time through the loop to ensure it is a legitimate floating-point value.

While the `proclaim` declaration does allow the user to write dangerous code, it is at least a reasonably principled loophole. `Proclaim` red-flags dangerous assumptions. If the user can be guaranteed that only code marked with `proclaim` declarations can behave in undefined ways, debugging broken programs becomes much easier.

Finally, it might be worth considering a third declaration, `probably`. `(probably flonum? x)` is a hint to the compiler that `x` is most likely a floating-point value, but could in fact be any type. Having a `probably` declaration can allow trace-scheduling compilers to pick good traces or optimistically open-code common cases.

7.5 Test Hoisting

Having one branch of a conditional test be the undefined effect `$` or `error` primop opens up interesting code motion possibilities. Let us call tests with an `(error)` arm "error tests," and tests with a `($)` arm "$-tests." These tests can be hoisted to earlier points in the code that are guaranteed to lead to the test. For example,

```
(block (print (+ x 3))
       (if (fixnum? x) (g x) ($)))
```

is semantically identical to

```
(if (fixnum? x) (block (print (+ x 3)) (g x))
    ($))
```

because the undefined effect operator can be defined to have any effect at all, including the effect of `(print (+ x 3))`. This can be useful, because hoisting type tests increases their coverage. In the example above, hoisting the `fixnum?` test allows the compiler to assume that `x` is a fixnum in the `(+ x 3)` code. Further, error and `$`-tests can be hoisted above code splits if the test is applied in both arms. For example, the type tests in

```
(if (> x 0)
    (if (pair? y) (bar) ($))
    (if (pair? y) (baz) ($)))
```

can be hoisted to a single type test:

```
(if (pair? y)
    (if (> x 0) (bar) (baz))
    ($))
```

Real savings accrue if loop invariant type tests get hoisted out of loops. For example, in a naive, declaration-free dot-product subroutine,

```
(define (dot-prod v w len)
  (do ((i (- len 1) (- i 1))
       (sum 0.0 (+ sum (* (vector-ref v i) (vector-ref w i)))))
      ((< i 0) sum)))
```

each time through the inner loop we must check that `v` and `w` have type vector. Since `v` and `w` are loop invariants, we could hoist the run-time type checks out of the loop, which would speed it up considerably. (In this particular example, we would have to duplicate the termination test `(< i 0)`, so that loop invariant code pulled out of the loop would only execute if the loop was guaranteed at least one iteration. This is a standard optimising compiler technique.)

Hoisting error tests requires us to broaden our semantics to allow for early detection of run-time errors. If execution from a particular control point is guaranteed to lead to a subsequent error test, it must be allowed to perform the error test at the control point instead.

In the general case, error and `$`-test hoisting is a variant of very-busy expression analysis [Dragon]. Note that hoisting `$`-tests gives a similar effect to the backwards type inferencing of [Kaplan]. Finding algorithms to perform this hoisting is an open research problem.

7.6 Other Applications

The general Reflow approach to solving quantity-based analyses presented in this paper can be applied to other data-flow problems in higher-order languages. The range analysis discussed in subsection 7.3 is a possible candidate for this type of analysis. Copy propagation in Scheme is also amenable to a Reflow-based solution.

A final example very similar to type recovery is future analysis. Some parallel dialects of Scheme [Mul-T] provide *futures,* a mechanism for introducing parallelism into a program. When the form `(future <exp>)` is evaluated, a task is spawned to evaluate the expression `<exp>`. The `future` form itself immediately returns a special value, called a *future*. This future can be passed around the program, and stored into and retrieved from data structures until its actual value is finally required by a “strict” operator such as `+` or `car`. If the future’s task has completed before the value is needed, the strict operation proceeds without delay; if not, the strict operator must wait for the future’s task to run to completion and return a value.

Futures have a heavy implementation expense on stock hardware, because all strict operators must check their operands. Future checking can add a 100% overhead to the serial run time of an algorithm on stock hardware.

“Future analysis” is simply realising that references to a variable that happen after the variable is used as an argument to a strict operator can assume the value is a non-future, because the strict operator has forced the value to resolve itself. Thus, in the lambda

```
(λ (x) (print (car x)) ...  (f (cdr x)) ...)
```

the `(cdr x)` operation can be compiled without future checking. Clearly, this is identical to the type-recovery analysis presented in this paper, and the same techniques apply.

8 Related Work

The method of non-standard abstract semantic interpretations has been applied to a variety of program analyses [Cousot] [Pleban] [RefCount] [Harrison] [CFASem]. A useful collection is [Abramsky]. The semantic basis for Scheme control-flow analysis, first discussed in [CFASem] and then in [Diss], also forms the basis for the type recovery semantics described here.

Steenkiste's dissertation [Steenkiste] gives some idea of the potential gains type recovery can provide. For his thesis, Steenkiste ported the PSL Lisp compiler to the Stanford MIPS-X processor. He implemented two backends for the compiler. The "careful" backend did full run-time type checking on all primitive operations, including `car`'s, `cdr`'s, vector references, and arithmetic operations. The "reckless" backend did no run-time type checking at all. Steenkiste compiled about 11,500 lines of Lisp code with the two backends, and compared the run times of the resulting executables. Full type checking added about 25% to the execution time of the program.

Clearly, the code produced by a careful backend optimised with type-recovery analysis will run somewhere between the two extremes measured by Steenkiste. This indicates that the payoff of compile-time optimisation is bounded by the 25% that Steenkiste measured. Steenkiste's data, however, must be taken only as a rough indicator. In Lisp systems, the tiny details of processor architecture, compiler technology, data representations and program application all interact in strong ways to affect final measurements. Some examples of the particulars affecting his measurements are: his Lisp system used high bits for type tags; the MIPS-X did not allow `car` and `cdr` operations to use aligned-address exceptions to detect type errors; his 25% measurement did not include time spent in the type dispatch of generic arithmetic operations; his generic arithmetic was tuned for the small integer case; none of his benchmarks were floating-point intensive applications; his measurements assumed interprocedural register allocation, a powerful compiler technology still not yet in common practice in Lisp and Scheme implementations; and Lisp requires procedural data to be called with the `funcall` primop, so simple calls can be checked at link time to ensure they are to legitimate procedures.

These particulars of language, hardware, implementation, and program can bias Steenkiste's 25% in both directions (Steenkiste is careful to discuss most of these issues himself). However, even taken as a rough measurement, Steenkiste's data do indicate that unoptimised type-checking is a significant component of program execution time, and that there is room for compile-time optimisation to provide real speed-up.

The idea of type recovery for Scheme is not new. Vegdahl and Pleban [Screme] discuss the possibility of "tracking" types through conditionals, although this was never pursued. The ORBIT compiler [ORBIT] is able to track the information determined by conditional branches, thus eliminating redundant tests. ORBIT, however, can only recover this information over trees of conditional tests; more complex control and environment structures, such as loops, recursions, and joins block the analysis.

Curtis discusses a framework for performing static type inference in a Scheme variant [Curtis], along the lines of that done for statically typed polymorphic languages such as ML [ML] or LEAP [Leap]. However, his work assumes that most "reasonable" Scheme programs use variables in a way that is consistent with a static typing discipline. In essence, Curtis' technique types variables, whereas the type recovery presented in this paper types variable references, an important distinction for Scheme. Note that without introducing type-conditional primitives that bind variables, and (perhaps automatically) rewriting Scheme code to employ these primitives, this approach cannot recover the information determined by conditional branches on type tests, an important source of type information.

[Tenenbaum] and [Kaplan] are typical examples of applying data-flow analysis to recover type information from latently-typed languages. The technique is also covered in chapter 10 of [Dragon]. These approaches, based on classical data-flow analysis techniques, differ from the technique in this paper in several ways:

- First, they focus on side-effects as the principle way values are associated with variables. In Scheme, variable binding is the predominant mechanism for associating values with variables, so the Scheme type recovery analysis must focus on variable binding.
- Second, they assume a fixed control-flow graph. Because of Scheme's first-class procedures, control-flow structure is not lexically apparent at compile time. The use of a CPS-based internal representation only makes this problem worse, since all transfers of control, including sequencing, branching, and loops are represented with procedure calls. The analysis in this paper handles procedure calls correctly.
- Third, they assume a single, flat environment. Scheme forces one to consider multiple bindings of the same variable. The reflow semantics of Section 5 correctly handles this complexity.
- Finally, they are not semantically based. The type recovery analysis in this paper is based on the method of non-standard abstract semantic interpretations. This establishes a formal connection between the analysis and the base language semantics. Grounding the analysis in denotational semantics allows the possibility of proving various useful properties of the analysis, although such proofs are beyond the scope of this paper.

These differences are all connected by the centrality of lambda in Scheme. The prevalence of lambda is what causes the high frequency of variable binding. Lambda allows the construction of procedural data, which in turn prevent the straightforward construction of a compile-time control-flow graph. Lambda allows closures to be constructed, which in turn provide for multiple extant bindings of the same variable. And, of course, the mathematical simplicity and power of lambda makes it much easier to construct semantically-based program analyses.

In the lambda operator, all three fundamental program structures — data, control, and environment — meet and intertwine. Thus, any analysis technique for Scheme must be prepared to face the three facets of lambda. In essence, the analysis in this paper is the application of classical data-flow type-recovery analysis in the presence of lambda.

9 Acknowledgements

I am very grateful to the people who helped me with this paper. My advisor, Peter Lee, substantially improved the early drafts. Norman Adams, Peter Lee, Jonathan Rees, and Mark Shirley reviewed the penultimate draft under extreme time pressure, and provided me with detailed, thoughtful comments and suggestions. Daniel Weise and an anonymous referee suggested some valuable references.

John Reynolds helped pin down the details of the semantics. Jonathan Rees pointed out to me the idea of representing user declarations as conditional branches in 1984. Uwe Pleban pointed out to me the type constraints that safe primop applications place on their arguments.

References

[Abramsky] Abramsky, Samson, and Hankin, Chris (ed.). *Abstract interpretation of declarative languages.* Ellis Horwood (1987).

[Bulldog] Ellis, John. *Bulldog: A Compiler for VLIW Architectures.* Ph.D. dissertation, Yale University. MIT Press (1986). Also available as Research Report YALEU/DCS/RR-364.

[CFASem'] Shivers, Olin. *The Semantics of Scheme Control-Flow Analysis (Prelim).* Technical Report ERGO-90-090, CMU School of Computer Science, Pittsburgh, Penn. (November 1988).

[CFASem] Shivers, Olin. *The Semantics of Scheme Control-Flow Analysis. (In preparation)*

[CFlow] Shivers, Olin. "Control Flow Analysis in Scheme." *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation* (June 1988). (Also available as Technical Report ERGO-88-60, CMU School of Computer Science, Pittsburgh, Penn.)

[Cousot] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." *4th ACM Symposium on Principles of Programming Languages* (1977), pp. 238–252.

[Curtis] Curtis, Pavel. *Constrained Quantification in Polymorphic Type Analysis.* Ph.D. dissertation, Cornell University, 1990. *(In preparation)*

[Diss] Shivers, Olin. *Control-Flow Analysis in Higher-Order Languages.* Ph.D. dissertation, CMU. *(In preparation)*

[Dragon] Aho, Sethi, Ullman. *Compilers, Principles, Techniques and Tools.* Addison-Wesley (1986).

[Fisher] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. "Parallel Processing: A Smart Compiler and a Dumb Machine," Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, *SIGPLAN Notices Vol. 19, No. 6* (June, 1984), pp. 37–47.

[Fixpoints] Young, Jonathan and Hudak, Paul. *Finding Fixpoints on Function Spaces.* Research Report YALEU/DCS/RR-505, Yale University (December 1986).

[Harrison] Harrison, Williams Ludwell III. "The Interprocedural Analysis and Automatic Parallelization of Scheme Programs." *Lisp and Symbolic Computation* vol. 2(3/4) (October 1989), pp. 179–396.

[Kaplan] Kaplan, Marc A., Ullman, Jeffrey D. "A Scheme for the Automatic Inference of Variable Types," *JACM* 27:1 (January 1980), 128–145.

[Leap] Pfenning, Frank and Lee, Peter. "Metacircularity in the Polymorphic Lambda-Calculus." *Theoretical Computer Science* (1990) *(To appear).*

[LispM] Moon, David. “Architecture of the Symbolics 3600.” *Proc. 12th Symp. on Computer Architecture*, Boston, MA (June 1985).

[ML] Milner, Robin. “The Standard ML Core Language.” *Polymorphism, vol. II(2)*. (October 1985). *Also* ECS-LFCS-86-2, University of Edinburgh, Edinburgh, Scotland (March 1986).

[MLComp] Appel, Andrew, and Jim, Trevor. “Continuation-passing, Closure-passing Style.” *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages* (January 1989), pp. 293–302.

[Mul-T] Kranz, D., Halstead, R., and Mohr, E. “Mul-T: A High-Performance Parallel Lisp” *Proceedings of SIGPLAN ’89 Symposium on Programming Languages Design and Implementation* (June 1989).

[ORBIT] Kranz, David, *et al.* “ORBIT: An Optimizing Compiler for Scheme.” *Proceedings of SIGPLAN ’86 Symposium on Compiler Construction* ACM (June 1986), pp. 219–233. Proceedings published as *SIGPLAN Notices* 21(7), July 1986.

[Pleban] Pleban, Uwe. *Preexecution Analysis Based on Denotational Semantics.* Ph.D. dissertation, University of Kansas, 1981.

[R3RS] Rees, J. and Clinger, W., editors. “The revised^3 report on the algorithmic language Scheme.” In *ACM SIGPLAN Notices* 21(12) (December 1986).

[Rabbit] Guy L. Steele. *Rabbit: A Compiler for Scheme.* AI-TR-474. MIT AI Lab (Cambridge, May 1978).

[Range] Harrison, William H. “Compiler Analysis of the Value Ranges for Variables,” *IEEE Transactions on Software Engineering* vol. SE-3, no. 3 (May 1977), pp. 243–250.

[RefCount] Hudak, Paul. “A Semantic Model of Reference Counting and its Abstraction.” *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming* (August 1986).

[Screme] Vegdahl, Steve and Pleban, Uwe. “The Runtime Environment for Screme, a Scheme Implementation on the 88000.” *ASPLOS-III Proceedings* (April 1989), pp. 127–182.

[SPARC] Garner, R., *et al.* “Scaleable processor architecture (SPARC).” COMPCON, IEEE (March, 1988) San Francisco, pp. 278–283.

[Spur] Hill, Mark *et al.* “Design Decisions in Spur.” *COMPUTER* (November 1986), pp. 8–22.

[Steenkiste] Steenkiste, Peter. *Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization.* Ph.D. dissertation, Stanford University (March, 1987). Computer Systems Laboratory, Technical report CSL-TR-87-324.

[Strict] Hudak, Paul and Bloss, Adrienne. “Variations on Strictness Analysis.” *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (August 1986).

[T] Rees, J. and Adams, N. "T: a dialect of LISP, or, Lambda: the ultimate software tool." Symposium on Lisp and Functional Programming, ACM (August 1982), pp. 114–122.

[Tenenbaum] Tenenbaum, A. *Type determination for very high level languages*. Rep. NSO-3, Courant Inst. Math. Sci., New York U., New York (1974).

[TMan] Rees, J., Adams, N. and Meehan, J. *The T Manual*. 4th edition, Yale University Computer Science Department (January 1984).

Notes

$\{$**Note Gensym**$\}$
The semantic function *nb* is, as presented, not a properly defined function, since it has hidden internal state. This is simple to remedy by providing an extra argument to all the main semantic functions that contains the entire set of contours currently allocated. This argument can be presented to *nb*, allowing it to choose some unused contour *b* not in this set. The new contour *b* is added to this set of allocated contours, which can then be passed forwards along the computational path to prevent *b* from ever being reallocated.

This addition to the semantics is trivial, and is omitted to simplify the presentation. In the intuitive model of a semantics as a functional interpreter in disguise, just think of *nb* as the Lisp `gensym` procedure, guaranteed to return a new value each time you call it.

$\{$**Note No Bottom**$\}$
One of the pleasant features of CPS Scheme is the scarcity of bottom values. Most of the semantic structures are unordered sets instead of CPOs. For example, the set D does not require a bottom value because all the expressions that can appear in a procedure call — constants, variable references, primops, and lambdas — are guaranteed to terminate when evaluated. In other words, the $\mathcal{A}_v$ function of subsection 4.3 never produces a bottom value. In the standard semantics, bottom can only show up as the final value for the entire program, never at an intermediate computation. For this reason, the disjoint union constructor + is taken to be a *set* constructor, not a domain constructor — it does not introduce a new bottom value. A careful treatment of the semantics of CPS Scheme at this level is beyond the scope of this paper; for further details, see [CFASem] or [Diss].

$\{$**Note Non-circular** `letrec`$\}$
It is an interesting curiosity that the definition of `letrec` presented here does not involve a recursive construction. Lambdas are closed over contour environments but not the variable environment, which is a global structure. So the actual evaluation of the `letrec`'s lambdas, $\mathcal{A}_v\, l_i\, \epsilon'\, ve = \langle l_i, \epsilon' \rangle$, is completely independent of the *ve* argument. The variable environment is not used because no variables are looked up in the evaluation of a lambda. We can close the lambdas over the new contour environment ϵ' without actually having the new contour's values in hand. This artifact of the factored semantics is considered in more detail in [CFASem].

$\{$**Note Recovering Continuations**$\}$
The reader may have noticed that the `+` primop is missing an opportunity to recover some available type information: it is not recovering type information about its continuation. For example, in code executed after the call to `+`'s continuation, we could assume that the quantity called has type *type/proc*. This information is not recovered because it isn't necessary. Since CPS Scheme is an intermediate representation for full Scheme, the user cannot write CPS-level continuations. All the continuations, variables bound to continuations, and calls to continuations found in the CPS Scheme program are introduced by the CPS converter. It is easy for the converter to mark these forms as it introduces them. So the types of continuation variables can be inferred statically, and there's no point in tracking them in our type recovery semantics.

{**Note Recovering Primops**}
In recovering information about the arguments to primops, we are essentially using information from “hidden conditional tests” inside the primop. The semantics of a (dangerous) CPS `car` primop is:

```
(define (car p cont)
  (if (pair? p) (cont (%car p))
      ($)))
```

where the subprimitive operation `%car` is only defined over cons cells, and `($)` is the “undefined effect” primop. Computation proceeds through the continuation `cont` only in the then arm of the type test, so we may assume that `p`’s value is a cons cell while executing the continuation. Recovering the type information implied by `car` reduces to recovering type information from conditional branches.

Of course, the compiler does not need to emit code for a conditional test if one arm is `($)`. It can simply take the undefined effect to be whatever happens when the code compiled for the other arm is executed. This reduces the entire `car` application to the machine operation `%car`.

A safe implementation of Scheme is one that guarantees to halt the computation as soon as a type constraint is violated. This means, for example, replacing the `($)` form in the else arm of the `car` definition with a call to the run-time error handler:

```
(define (car p cont)
  (if (pair? p) (cont (%car p))
      (error)))
```

Of course, type information recovered about the arguments to a particular application of `car` may allow the conditional test to be determined at compile time, again allowing the compiler to fold out the conditional test and error call, still preserving type-safety without the run-time overhead of the type check.

{**Note Run-time Errors**}
One detail glossed over in the functional definition of closures and other parts of the PTREC semantics is the handling of run-time errors, *e.g.*, applying a two-argument procedure to three values, dividing by zero, or applying `+` to a non-number. This is simple to remedy: run-time errors are defined to terminate the program immediately and return the current type cache. The extra machinery to handle these error cases has been left out of this paper to simplify the presentation; restoring it is a straightforward task.

{**Note VLIW**}
VLIW’s could be ideal target machines for languages that require run-time typechecking. For example, when compiling the code for a safe `car` application, the compiler can pick the trace through the type test that assumes the `car`’s argument is a legitimate pair. This will almost always be correct, the sort of frequency skew that allows trace scheduling to pay off in VLIW’s. The actual type check operation can percolate down in the main trace to a convenient point where ALU and branch resources are available; the error handling code is off the main trace.

Common cases for generic arithmetic operations are similarly amenable to trace picking. The compiler can compile a main fixnum trace (or flonum trace, as the common case may be), handling less frequent cases off-trace. The overhead for bignum, rational, and complex arithmetic ops will dominate the off-trace time in any event, whereas the lightweight fixnum or flonum case will be inlined.

The VLIW trace-scheduling approach to run-time type safety has an interesting comparison to the automatic tag checking performed by Lisp Machines. Essentially, we have taken the tag-checking ALU and branch/trap logic, promoted it to general-purpose status, and exposed it to the compiler. These hardware resources can now be used for non-typechecking purposes when they would otherwise lay idle, providing opportunities for increased fine-grained parallelism.

The Lisp Machine approach is the smart hardware/fast compiler approach; the VLIW approach is the other way ’round.

Notes