Compile-Time Garbage Collection in Typed Smalltalk

Carl McConnell
Ralph E. Johnson
University of Illinois at Urbana-Champaign[1]

This paper describes how Typed Smalltalk can be augmented with an *effect system* that will allow the TS compiler to determine when objects may be allocated on the stack rather than the heap, thus reducing and in many cases eliminating the need for run-time garbage collection.

1 Introduction

Languages with automatic storage management—"garbage collection"—make programming simpler and less error-prone, since the programmer needn't be concerned with storage deallocation, dangling pointer errors, or any of the other issues that arise with explicitly managed storage. However, the implementation of such a language can leave something to be desired as far as performance is concerned: the sporadic pauses of unpredictable duration that occur due to garbage collection can render the language useless for some applications. This is especially likely to be true for real-time applications.

One way to reduce the cost of garbage collection is to allocate storage on the stack, where it can be cheaply reclaimed, rather than on the heap. Of course, the compiler must guarantee that storage allocated on the stack will not be deallocated until it is no longer in use. Thus, it must analyze programs to determine when to allocate on the stack rather than on the heap.

This paper describes how the compiler for Typed Smalltalk (TS) [JGZ88] [GJ90] can perform such an analysis using an *effect system* [Luc87] [LG88]. An effect system is analogous to a type system, and works in conjunction with it. However, rather than describing the values of expressions, an effect system describes the side-effects of expressions. The compiler can make use of this information to perform compile-time garbage collection.

This paper assumes that the reader has some familiarity with TS.

2 Types and Effects in TS

A type and effect system has three basic entities: *types*, which describe the values of expressions; *effects*, which describe the side-effects of expressions; and *regions*, which describe the areas of memory in which side-effects occur. Types, effects, and regions are collectively referred to as *descriptions*, and are interrelated:

[1]Authors' address: Department of Computer Science, University of Illinois, 1304 W. Springfield, Urbana, IL 61801 USA. *E-mail*: `johnson@cs.uiuc.edu, mcconnel@cs.uiuc.edu`. *Phone*: (217) 244–0093

- The type of a method or block includes a *latent effect*, which describes the side-effects that the method or block will have when it is applied.

- The type of an expression may include a region indicating where the storage for the result of that expression is allocated.

- Effect descriptions include a region indicating the area of memory affected.

A region describes a set of memory locations. For example, a region could represent the set of memory locations with even addresses. Distinct regions describe disjoint sets of locations. An example of a region description is *#even*, a region constant that might be used to represent the set of locations just mentioned. A more relevant example is the *#static* region, which we will use to denote that portion of memory containing objects that exist throughout the lifetime of a Smalltalk program, such as the objects true and false. Arbitrary regions may be indicated using variables.

An effect represents a side-effect on some region of memory. There are four kinds of effects: *reads:*, *writes:*, *allocs:*, and *pure*. The first three respectively describe the effect of reading, writing, or allocating storage in a given region; the last describes the absence of side-effects. Every effect description includes the region of memory where the effect occurs. Effects can be combined using the union constructor +. Examples of effect descriptions are *<reads: #even>*, *<(reads: #a) + (writes: #b) + (allocs: #c)>*, and *<(writes: #region) + pure>*, which is the same as *<writes: #region>*. Arbitrary effects may be indicated using variables.

A type describes the value an expression computes. The type system of TS is discussed fully in [Gra89], so only the extensions made to accommodate the effect system will be mentioned here. One such extension allows a type to indicate the region in which a value is allocated. For example, the type *<SmallInteger in: #static>* describes a SmallInteger allocated in the *#static* region of memory, which is where all SmallIntegers are considered to be "allocated." Similarly, the type *<OrderedCollection of: SmallInteger in: R>* describes an OrderedCollection of SmallIntegers allocated in an arbitrary region R.

The second extension to the type system allows the latent effect of a block or method to appear in its type. For example, *<Block of: SmallInteger returns: (OrderedCollection of: SmallInteger in: R) effect: (writes: R)>* describes a block that takes a SmallInteger as argument, returns an OrderedCollection of SmallIntegers in region R, and has the side-effect of writing in region R.

The static analysis of a program involves determining descriptions:

- Every variable is described by a type and a region. The region of a variable indicates where the storage for that variable is allocated, and is implicitly determined by the kind of the variable (instance variable, temporary, etc.).

- Every expression is described by the type of its result and the effect of its evaluation.

- Every method and block is described by a type and a latent effect.

Descriptions are computed using inference rules introduced in subsequent sections.

Given this machinery, compile-time garbage collection is simple:

```
SequenceableCollection subclass: #OrderedCollection
    instanceVariables: 'firstIndex <SmallInteger in: #static>
                       lastIndex <SmallInteger in: #static>'
    classVariables: ''
    typeParameters: 'ElementType'
    poolDictionaries: ''
    category: 'Sequenceable-Collections'
```

Figure 1: Class definition for OrderedCollection.

Compile-time Garbage Collection and Effect Masking Rule. Consider a message send expression E and a region ρ. If ρ does not appear in the type of E or the types of the free variables of E, then any effect of E on ρ can be *masked* (replaced by the effect *pure*). Furthermore, any storage associated with ρ can be deallocated after E returns.

This rule follows from the fact that the region ρ is inaccessible after E returns, which places an upper bound on the lifetime of the region. For a region satisfying the conditions of the rule, the compiler can generate code that explicitly allocates and deallocates the associated storage. Note, however, that this storage can be managed using stack allocation only if its size is known at compile-time. If it is not, the storage must be managed using other techniques, such as maintaining a linked list of used storage blocks to be deallocated when the expression returns.

An effect system is a powerful tool for reasoning about programs. Its uses include:

- Code parallelization. For example, two expressions can be evaluated concurrently if they do not interfere (read or write the same region). The notion of an effect system was developed with this purpose in mind [Luc87].
- Code optimization. An example of this is constant folding, where a message send with no side-effects and constant arguments is evaluated at compile-time.

Although we expect the TS effect system to eventually be employed for some of these purposes, this paper discusses only compile-time garbage collection.

3 Using Types and Effects

Class declarations differ from those normally used in TS only in that the type declarations of instance variables may include a region. By way of example, the class declaration for OrderedCollection is shown in Figure 1. However, the type declaration *<SmallInteger in: #static>* could have been abbreviated to simply *<SmallInteger>*, since SmallIntegers are always “allocated” in the *#static* region.

```
{ localTypeVariables: P1 P2
  localEffectVariables: E1 E2
  localRegionVariables: R1 R2
  receiverType: <Self>
  arguments: arg1 <arg1Type> arg2 <arg2Type>
  temporaries: temp <tempType>
  blockArguments: blkArg <blkArgType>
  effect: <reads: SelfRegion>
  returnType: <Self> }

message selector and argument names
    "comment stating purpose of message"

    | temporary variable names |
    statements
```

Figure 2: Template for typed methods.

A typed method template is shown in Figure 2. In addition to the usual TS method template fields, the template has localEffectVariables: and localRegionVariables: fields, whose arguments are capitalized identifiers representing local description variables; and an effect: field, whose argument is an effect description. Local description variables are instantiated during description checking to descriptions determined according to the call site. The default effect given in the template makes use of the region variable *SelfRegion*, which denotes the region in which self is allocated.

Let us consider how the effect of the do: method of OrderedCollection, shown in Figure 3, can be inferred from the method itself. First, the temporaries are automatically assigned the region *#temporary*. The effects of the two assignments to these temporaries are respectively *writes: #temporary + pure* and *writes: #temporary + reads: SelfRegion*. (The *reads: SelfRegion* effect is due to the expression self size.) The effect *writes: #temporary* can be replace by *pure*, since this effect isn't visible outside the method. Thus, the total effect of the first two statements in the method is *reads: SelfRegion*. The block receiving the whileTrue: message also clearly has effect *pure*. However, the block that is the argument to this message has two effects. One is due to the evaluation of aBlock, and it is this effect that is denoted by the effect variable E. The other effect is *reads: SelfRegion*, which is due to the at: message sent to self. The effect of the method as a whole is simply the union of the effects of the statements in it. Keeping in mind that the effect *pure* acts as the identity element for the union operation, we arrive at the effect shown in the declaration.

As another example, the select: method of OrderedCollection is shown in Figure 4. In this method, a local region variable R denotes the region in which storage is allocated by the new method. The effect of the new: method is *allocs: Q*. From the declaration for newCollection,

```
{ localTypeVariables: P
  localEffectVariables: E
  arguments: aBlock <Block of: ElementType returns: P effect: E>
  temporaries: index <SmallInteger> length <SmallInteger>
  effect: <(reads: SelfRegion) + E>
  returnType: <Self> }

do: aBlock
    "Evaluate aBlock for each of the receiver's elements."

    | index length |
    index ← 0.
    length ← self size.
    [(index ← index + 1) <= length]
        whileTrue: [aBlock value: (self at: index)]
```

Figure 3: The do: method for OrderedCollection.

```
{ localRegionVariables: R
  arguments: aBlock <Block of: ElementType returns: (True + False) effect: pure>
  temporaries: newCollection <OrderedCollection of: ElementType in: R>
  effect: <(reads: SelfRegion) + (allocs: R) + (writes: R)>
  returnType: <OrderedCollection of: ElementType in: R> }

select: aBlock
    "Answer a collection containing the elements for which aBlock is true."
    | newCollection |
    newCollection ← OrderedCollection new.
    self do: [:each | (aBlock value: each)
        ifTrue: [newCollection add: each]].
    ↑newCollection
```

Figure 4: The select: method for OrderedCollection.

the compiler can infer that $Q = R$, and therefore that the type of the right-hand side of the first statement is *OrderedCollection of: ElementType in: R*, and that its effect is *allocs: R*. We have already seen that the do: method has effect *reads: SelfRegion* in addition to the effect of evaluating its block argument, which in this case is *writes: R*. The effect shown in

	t ::=	Type ::=
(object type)	$C(\rho)$	$\mathit{ClassName}$ ($\underline{\mathsf{in:}}$ Region)
	$\mid C(\rho, \vec{t}\,)$	$\mid$ $\mathit{ClassName}$ ($\underline{\mathsf{of:}}$ $\mathit{Type})^{+}$ ($\underline{\mathsf{in:}}$ Region)
(block type)	$\mid \vec{t} \xrightarrow{\epsilon} t$	$\mid$ $\underline{\mathsf{Block}}$ ($\underline{\mathsf{of:}}$ $\mathit{Type})^{*}$ $\underline{\mathsf{returns:}}$ Type $\underline{\mathsf{effect:}}$ Effect
(union type)	$\mid t + t$	$\mid$ Type $\underline{+}$ Type
	$\mid (t)$	$\mid$ $\underline{(}\mathit{Type}\underline{)}$
(type variable)	$\mid p$	$\mid P$
($\top$)	$\mid \mathit{Anything}$	$\mid$ $\underline{\mathsf{Anything}}$
($\bot$)	$\mid \mathit{Nothing}$	$\mid$ $\underline{\mathsf{Nothing}}$

Figure 5: Abstract and concrete syntax of types.

the declaration is the union of these various effects.

To see how opportunities for compile-time garbage collection may arise, consider the expression

(oc select: [:each | each even]) size

where oc has type *OrderedCollection of: SmallInteger in: Q*. The effect of this expression is *(reads: Q) + (allocs: R) + (writes: R)*, since this is the effect of the select: method. However, the region R meets the criteria for effect masking and compile-time garbage collection, and so the effect can be collapsed to *reads: Q*. The compiler can generate code to allocate storage for region R before the expression is evaluated, and the location of this storage can be passed to select: and then to new: using some run-time convention. After the expression is evaluated, the storage allocated for R can be deallocated.

4 Type and Effect System Specification

The description of the type and effect system presented in this section is based on that of the type system given in [Gra89], with which some familiarity is assumed. It focuses on deviations from, and additions to, that type system.

4.1 Syntax

The abstract and concrete syntax for type expressions is shown in Figure 5, that for effect expressions in Figure 6, and that for region expressions in Figure 7. The abstract syntax is on the left and concrete syntax is on the right. In the concrete syntax, grammar terminals are underlined, (something)* represents zero or more repetitions of the something, and (something)$^+$ represents one or more repetitions of the something.

We use the abstract syntax in inference rules and the concrete syntax in examples. Type expressions are denoted by s, t, and u, effect expressions by ϵ, and region expressions by ρ.

	ϵ ::=	*Effect* ::=
(read effect)	$Read(\rho)$	(reads: *Region*)
(write effect)	$\mid Write(\rho)$	| (writes: *Region*)
(alloc effect)	$\mid Alloc(\rho)$	| (allocs: *Region*)
(union effect)	$\mid \epsilon + \epsilon$	| *Effect* + *Effect*
	$\mid (\epsilon)$	| (*Effect*)
(effect variable)	$\mid f$	| *E*
(empty effect)	$\mid pure$	| pure

Figure 6: Abstract and concrete syntax of effects.

	ρ ::=	*Region* ::=
(region constant)	m_{Name}	#*Name*
(region variable)	$\mid r$	| *R*

Figure 7: Abstract and concrete syntax of regions.

Type variables are denoted by p (abstract syntax) or by P (concrete syntax), effect variables by f (abstract) or by E (concrete), and region variables by r (abstract) or by R (concrete). Lists (tuples) of types are denoted by $\vec{t} = < t_1, t_2, \ldots, t_n >$. The empty list is denoted by $<>$ and $t \cdot \vec{t}$ is the list $< t, t_1, t_2, \ldots, t_n >$. Similar notation is used for lists of type variables. C denotes a class name and m denotes a message selector.

4.2 Inclusion

The intuitive meaning of the inclusion relation $\sqsubseteq$ is that if $s \sqsubseteq t$, then everything of description s is also of description t. The inclusion rules for effects and regions are:

- $\rho' \sqsubseteq \rho$ iff $\rho' = \rho$
- $\epsilon'_1 + \cdots + \epsilon'_n \sqsubseteq \epsilon_1 + \cdots + \epsilon_n$ iff $\forall i \exists j . \epsilon_j = \epsilon'_i$

For types, the following inclusion rules hold:

- $C'(\rho', T') \sqsubseteq C(\rho, T)$ iff $C' = C$, $\rho' = \rho$, and $T' = T$
- $\vec{s} \xrightarrow{\epsilon} t \sqsubseteq \vec{s}' \xrightarrow{\epsilon'} t'$ iff $\vec{s}' \sqsubseteq \vec{s}$, $t \sqsubseteq t'$, and $\epsilon \sqsubseteq \epsilon'$

Other inclusion rules are as described in [Gra89].

4.3 Description Checking

Description checking is presented in terms of inference rules. Besides a set of description constraints $\mathcal{C}$, we need a set of assumptions $\mathcal{A}$ about the types of variables and the regions in which they are located. The notation $\mathcal{C}, \mathcal{A} \vdash e : t$ means that from the constraints $\mathcal{C}$ and assumptions $\mathcal{A}$ we can infer that an expression e has type t, $\mathcal{C}, \mathcal{A} \vdash e \,!\, \epsilon$ means we can infer that e has effect ϵ, and $\mathcal{C}, \mathcal{A} \vdash v@\rho$ means we can infer that variable v is located in region ρ. The notation $\mathcal{A}.v : t$ means to extend the set $\mathcal{A}$ with the assumption that variable v has type t. A horizontal bar represents logical implication: if we can infer what is above it, then we can infer what is below it.

The type of a variable is known from its declaration, its region is trivially inferred from its kind, and it has a read effect on that region:

$$\mathcal{C}, \mathcal{A}.v : t \vdash v : t$$

$$\vdash v@\rho$$

$$\vdash v \,!\, \mathit{Read}(\rho)$$

Global variables are in the region $\#\mathit{static}$, instance variables are in $\mathit{SelfRegion}$, and temporaries are in $\#\mathit{temporary}$.

The types of the pseudo-variables true, false, and nil are constants:

$$\vdash \underline{\mathsf{true}} : \mathit{True}(m_{static})$$

$$\vdash \underline{\mathsf{false}} : \mathit{False}(m_{static})$$

$$\vdash \underline{\mathsf{nil}} : \mathit{UndefinedObject}(m_{static})$$

Since pseudo-variables are variables, they have no effect.

The type of a literal (constant) is inferred from its class. If this type is not general enough then an explicit type declaration can be used to supply the "correct" type [Gra89].

$$\vdash n : \mathit{Integer}(m_{static})$$

$$\vdash \underline{\$}c : \mathit{Character}(m_{static})$$

Literals have no effect:

$$\vdash n \,!\, \mathit{pure}$$

$$\vdash \underline{\$}c \,!\, \mathit{pure}$$

The type of an assignment statement $v \leftarrow e$ is the type of e. Its effect is the union of the effect of evaluating e and that of writing in the region of v:

$$\frac{\begin{array}{c} \mathcal{C}, \mathcal{A} \vdash v : s \qquad \mathcal{C}, \mathcal{A} \vdash v@\rho \\ \mathcal{C}, \mathcal{A} \vdash e : t \qquad \mathcal{C}, \mathcal{A} \vdash e \,!\, \epsilon \\ \mathcal{C} \vdash t \sqsubseteq s \end{array}}{\mathcal{C}, \mathcal{A} \vdash (v \leftarrow e) : t \qquad \mathcal{C}, \mathcal{A} \vdash (v \leftarrow e) \,!\, \mathit{Write}(\rho) + \epsilon}$$

The type of a statement sequence is the type of the last statement in the sequence. The effect of a statement sequence is the union of the effects of the individual statements:

$$\frac{\mathcal{C}, \mathcal{A} \vdash e_1 : t_1 \quad \ldots \quad \mathcal{C}, \mathcal{A} \vdash e_n : t_n \qquad \mathcal{C}, \mathcal{A} \vdash e_1 \,!\, \epsilon_1 \quad \ldots \quad \mathcal{C}, \mathcal{A} \vdash e_n \,!\, \epsilon_n}{\mathcal{C}, \mathcal{A} \vdash (e_1 \ldots e_n) : t_n \qquad \mathcal{C}, \mathcal{A} \vdash (e_1 \ldots e_n) \,!\, \epsilon_1 + \cdots + \epsilon_n}$$

The type and effect of a block is inferred from the declared types of its arguments and the inferred type and effect of its statement list:

$$\frac{\mathcal{C}, \mathcal{A}.\vec{y} : \vec{t} \vdash e : u \qquad \mathcal{C}, \mathcal{A}.\vec{y} : \vec{t} \vdash e \,!\, \epsilon}{\mathcal{C}, \mathcal{A} \vdash [\vec{y} \mid e] : \vec{t} \xrightarrow{\epsilon} u}$$

The inference rule for return statements requires some explanation. The only place a return statement can appear is as the last statement in the statement list of a method or a block. The types of the top-level statements in the statement list of a method can be ignored. The type of a block will always be checked for inclusion within some block type. To afford the maximum freedom to such blocks, the type given to return statements is the special type *Nothing*, which is included in any type. The set of assumptions $\mathcal{A}$ contains a special variable **returnValue** whose type is the declared return type of the method being type-checked. The effect of a return statement is simply the effect of its expression.

$$\frac{\mathcal{C}, \mathcal{A} \vdash \underline{\textsf{returnValue}} : t \qquad \mathcal{C}, \mathcal{A} \vdash e : t \qquad \mathcal{C}, \mathcal{A} \vdash e \,!\, \epsilon}{\mathcal{C}, \mathcal{A} \vdash (\uparrow e) : \mathit{Nothing} \qquad \mathcal{C}, \mathcal{A} \vdash (\uparrow e) \,!\, \epsilon}$$

Due to the simplicity of Smalltalk, the only remaining kind of expression is the message send. Before giving the inference rule for method sends, we need some auxiliary definitions. Let H be a hierarchy of typed class definitions and $H < C, m >$ be the definition in class C for method m, i.e.

$$H < C, m >= \vec{p} \sqsubseteq \vec{u}\; C(\rho, \vec{s}\,)\; m\; \vec{y} : \vec{t} \uparrow u \,!\, \epsilon\; \vec{z} : \vec{t}' e$$

where $\vec{p} \sqsubseteq \vec{u}$ are local type variables with range declarations, $C(\rho, \vec{s}\,)$ is the type of the receiver, $\vec{y} : \vec{t}$ are typed arguments, u is the return type, ϵ is the effect, $\vec{z} : \vec{t}'$ are the typed temporaries, and e is the method body. The notation

$$C(\rho, \vec{s}\,) \cdot \vec{t} \xrightarrow{\epsilon} u \mid \vec{p} \sqsubseteq \vec{u}$$

denotes a method type where $C(\rho, \vec{s})$ is the receiver type, $\vec{t}$ is the list of argument types, u is the return type, ϵ is the effect, and $\vec{p} \sqsubseteq \vec{u}$ are range constraints. For notational convenience, we use the following inference rule to map method definitions to method types:

$$\frac{H < C, m >= \vec{p} \sqsubseteq \vec{u}\; C(\rho, \vec{s}\,)\; m\; \vec{y} : \vec{t} \uparrow u \,!\, \epsilon\; \vec{z} : \vec{t}' e}{\vdash < C, m >_d : C(\rho, \vec{s}\,) \cdot \vec{t} \xrightarrow{\epsilon} u \mid \vec{p} \sqsubseteq \vec{u}}$$

The notation $\vdash < C, m >_d : t$ means it can be inferred that t is the declared method type of the method invoked when sending the message m to an instance of class C.

In the following inference rule for message sends, $e : C(\rho, \vec{s}\,)$ is the type of the receiver, $\vec{e} : \vec{t}$ and $\vec{e}\,!\,\vec{\epsilon}$ respectively describe the types and effects of the arguments, ϵ_i denotes a component of $\vec{\epsilon}$, and $e\ m\ \vec{e}$ denotes a message send:

$$\frac{\begin{array}{c} \mathcal{C}, \mathcal{A} \vdash e : C(\rho, \vec{s}\,) \quad \mathcal{C}, \mathcal{A} \vdash e : \epsilon \\ \mathcal{C}, \mathcal{A} \vdash \vec{e} : \vec{t} \quad \mathcal{C}, \mathcal{A} \vdash \vec{e}\,!\,\vec{\epsilon} \\ \mathcal{C}, \mathcal{A} \vdash < C, m >_d : C(\rho', \vec{s}\,'\,) \cdot \vec{t}\,' \xrightarrow{\epsilon'} u' \mid \vec{p} \sqsubseteq \vec{u} \\ \mathcal{C}.(\vec{p} \sqsubseteq \vec{u}).(C(\rho, \vec{s}\,) \sqsubseteq C(\rho', \vec{s}\,'\,)).(\vec{t} \sqsubseteq \vec{t}\,') \vdash (u' \sqsubseteq u) \wedge (\epsilon' \sqsubseteq \epsilon) \end{array}}{\mathcal{C}, \mathcal{A} \vdash (e\ m\ \vec{e}\,) : u \quad \mathcal{C}, \mathcal{A} \vdash (e\ m\ \vec{e}\,)\,!\,(\epsilon + \epsilon_1 + \cdots + \epsilon_n)}$$

An inference rule is also needed for method definitions. Recall that the abstract definition for a method m looks like

$$\vec{p} \sqsubseteq \vec{u}\ C(\rho, \vec{s}\,)\ m\ \vec{y} : \vec{t} \uparrow u\ \ !\ \epsilon\ \vec{z} : \vec{t}\,' e.$$

If, by adding type declarations appropriately to $\mathcal{C}$ and $\mathcal{A}$, it can be inferred that e has type u and effect ϵ, then it can be inferred that the definition of m is of type

$$C(\rho, \vec{s}\,) \cdot \vec{t} \xrightarrow{\epsilon} u \mid \vec{p} \sqsubseteq \vec{u}.$$

This is expressed in the following inference rule where $\mathsf{self} : C(\rho, \vec{s}\,)$ is the implicit type of the pseudo-variable self, and $\mathsf{returnValue}$ is a special variable used for type-checking return statements:

$$\frac{\begin{array}{c} H < C, m > = \vec{p} \sqsubseteq \vec{u}\ C(\rho, \vec{s}\,)\ m\ \vec{y} : \vec{t} \uparrow u\ \ !\ \epsilon\ \vec{z} : \vec{t}\,' e \\ \mathcal{C}.\vec{p} \sqsubseteq \vec{u},\ \mathcal{A}.\underline{\mathsf{self}} : C(\rho, \vec{s}\,).\ \vec{y} : \vec{t}.\ \underline{\mathsf{returnValue}} : u.\ \vec{z} : \vec{t}\,' \vdash e : u \end{array}}{\mathcal{C}, \mathcal{A} \vdash < C, m > : C(\rho, \vec{s}\,) \cdot \vec{t} \xrightarrow{\epsilon} u \mid \vec{p} \sqsubseteq \vec{u}}$$

Note that $< C, m >: t$ denotes the true inferred type of a method m for class C, which must be included in the declared method type $< C, m >_d : t'$.

5 Effects and Real-Time Programming

Although the techniques described thus far reduce the need for garbage collection, they do not always eliminate it: occasionally, so much storage may be requested that garbage collection is necessary to recover storage that has not yet been deallocated but is nevertheless garbage. For time-critical methods, garbage collection is unacceptable. We wish to be able to certify that a method is "real-time"; that is, that it will not cause the garbage collector to be invoked.

In general, the only way to guarantee that a method will not cause the garbage collector to be invoked is to ensure that the method does not allocate storage on the heap; in other words, all storage for the method must be allocated on the stack. Storage for a method can be stack- rather than heap-allocated if upper bounds on both the lifetime and the size of the storage are known. The effect system allows us (via the compile-time garbage collection rule) to find an upper bound on the lifetime. However, an upper bound on the amount must

```
{ arguments: aBlock <Block of: ElementType returns: (True + False) realTimeEffect: pure>
  realTimeEffect: <pure>
  returnType: <SmallInteger> }

count: aBlock
    "Answer the number of elements for which aBlock is true."
    | n |
    n ← 0.
    self do: [:each | (aBlock value: each)
            ifTrue: [n ← n + 1]].
    ↑n
```

Figure 8: A real-time method.

be found by some other means. In order to inform the compiler that both of these upper bounds *must* be found, we introduce the realTimeEffect: declaration for real-time methods, to be used instead of the effect: declaration. The realTimeEffect: declaration causes the compiler to ensure that all storage for the method can be allocated on the stack, and that all methods called are also real-time. An example of a real-time method is shown in Figure 8.

Another optimization that the effect system can be used for is "in-line substitution" of instance variables: replacing instance variables by the objects to which the instance variables refer. For example, class Rectangle has two instance variables referring to Points. After in-line substitution, these Points would be stored as part of instances of Rectangle rather than as separate objects. This optimization conserves storage by eliminating an instance variable and an object header. It may be performed on an instance variable I of a class C if the class of the object to which I refers is known so that the appropriate amount of storage can be allocated; and if the region of the object to which I refers does not appear free in any "public" method (a method which might be invoked by an object other than self) of C, thus ensuring that the object need not be represented in the conventional object format.

References

[GJ90] Justin Graver and Ralph Johnson. A type system for Smalltalk. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 136–150, 1990.

[Gra89] Justin Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages.* PhD thesis, University of Illinois at Urbana-Champaign, 1989.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An optimizing compiler for Smalltalk. In *Proceedings of OOPSLA '88*, pages 18–26, November 1988. printed as SIGPLAN Notices, 23(11).

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[Luc87] John M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1987.