CENTRE DE RECHERCHE EN INFORMATIQUE

CONTROL-FLOW EFFECTS FOR ESCAPE ANALYSIS

Yan-Mei TANG

Juin 1992

Workshop on Static Analysis 92 (JPASPEFL)
Bordeaux, France, September 23-25, 1992

Document EMP–CRI A-225

Control-Flow Effects for Escape Analysis

Yan-Mei Tang
Pierre Jouvelot

CRI, Ecole des Mines de Paris, France

Abstract

We present a static system that reconstructs the types and control-flow information of expressions in an implicitly typed functional language with imperative operations. Just as types describe the possible values of expressions, control-flow information describes the possible flow of control during evaluation. In functional languages, the control-flow information of an expression is defined as the set of all of the functions possibly called during its evaluation.

We introduce a static semantics for inferring types and control-flow information, and state its consistency with respect to the dynamic semantics. As a direct application of this analysis, we present a static criterion for identifying escaping functions, whose environments outlive their lexical scope, based on the inferred type and control-flow information. Non-escaping functions can be efficiently stack-allocated, leaving heap-allocation only to escaping functions.

1 Introduction

First-class function values are represented as closures by compilers. Closures are composed of two parts: the function code and the free variables that form the function environment. Finding an efficient allocation strategy for closure environments is therefore important for optimizing compilers of functional languages. Closures can be allocated either in the heap or in the stack.

Since functions are first-class values, they may outlive the environment in which they are defined; these so-called *escaping* functions do not obey the LIFO stack-allocation strategy and must be heap-allocated. Heap-allocation is more general than stack-allocation in the sense that heap-allocation can be used for all functions, while stack-allocation is only safe for non-escaping functions. However, stack-allocation is cheaper than heap-allocation because useless storage is simply reclaimed by updating a pointer instead of calling the garbage collector (but see [1] for a different point of view).

A good strategy for closure allocation for functional languages is to stack-allocate non-escaping functions while reserving the more expensive heap-allocation to escaping ones. A key problem is thus to identify safely and as precisely as possible escaping functions.

We present a new static analysis for identifying escaping functions in functional languages that support imperative constructs and separate compilation. This static analysis directly relies on a new effect system that infers the type and control-flow effect of expressions. This control-flow static semantics is defined within the type and effect framework [4,8,6].

Effect systems allow the compile-time determination of behavioral properties of programs such as side-effects [8,13], time complexity [3] or continuation effects [5]. Control-flow determination is difficult in higher-order languages since the function call graph is dynamic. By using the effect framework, we are able to get an approximation of this graph, while allowing separately compiled functions to be independently analyzed.

We discuss the related work (Section 2), present our language syntax and dynamic semantics (Section 3), describe our new control-flow effect system (Section 4), show its application to escape analysis (Section 5) and conclude (Section 6).

2 Related Work

Escaping functions can be identified either at compile time, based on a static analysis of programs, or at run time, using a run-time checking mechanism [2]. We only discuss compile-time approaches.

The Scheme [10] compilers Rabbit [12] and ORBIT [7] perform very simple escape analysis to optimize the closure allocation strategy. These analyses are syntax-based, i.e., the escaping functions are identified by their syntactical context through a recursive walk of expressions. In our approach, the escape analysis is based on the type and control-flow information of expressions, as computed by a type and effect inference system. Our escape analysis is thus more precise than that of Rabbit or ORBIT, in particular when dealing with higher-order functions.

In Shivers's thesis [11], the control-flow analysis is based on a non-standard abstract interpretation. This abstract interpretation is expensive and does not support separate compilation, which limits its application. By contrast, our control-flow analysis is performed by a type and effect inference system that supports the separate compilation of modules.

Control effects, defined in [5], are somewhat related to the control-flow information we gather here. However, these control effects are targeted to non-functional behaviors, such as those created by branches or continuations. Also, this analysis is targeted to an explicitly typed language, which allows explicit polymorphism.

The type and effect reconstruction algorithm used in [13] for side-effect analysis, together with its proof of the consistency of the analysis with respect to the dynamic semantics, can be easily applied to our static semantics.

3 Language Definition

Our language is a simple imperative extension of the lambda-calculus. Since operations on locations are of particular interest in identifying escaping functions, they are explicitly specified here:

`e` ::=	`x`	*value identifier*
	`(e e')`	*application*
	`(lambda n (x) e)`	*abstraction*
	`(let (x e) e')`	*lexical definition*
	`(new e)`	*initialization*
	`(get e)`	*dereference*
	`(set e e')`	*assignment*

Notice that all lambda expressions are explicitly given a name `n` (from the domain *Id* of identifiers) which is used to uniquely identify them. It is straightforward to transform a program with unnamed lambda expressions into this language.

The dynamic semantics is specified by a set of transition rules [9]. In addition to defining the usual evaluation process, this semantics also keeps track, in *traces*, of the functions called during the evaluation of expressions. Given a store s mapping locations to values and an environment E mapping variables to values, the dynamic semantics associates an expression `e` with the value v it computes, the trace f of the function names called during evaluation and the possibly updated store s'. We note:

$$s, E \vdash \texttt{e} \rightarrow v, f, s'$$

4 Type and Control-Flow Semantics

We specify a static semantics that approximates the control-flow behavior of expressions. It is presented as a set of inference rules that, for each expression, specify its type and the trace of the functions which may be called during evaluation.

A control c abstracts a trace f in the dynamic semantics and thus records all of the functions that can possibly be called during the evaluation of an expression. It can either be the constant $\emptyset$ which means that the expression is a basic block, a singleton $\{\texttt{n}\}$ in which `n` is a function name, a control variable *cv* or a set of function names, indicated by the infix union operator $\cup$.

A type t can either be the basic type *unit*, a type variable *tv*, a reference type $ref(t)$ which represents updatable locations containing values of type t (the reference type makes locations identifiable at compile-time), a function type $t \xrightarrow{c} t'$ where c is the latent control-flow information (a latent control-flow is the set of functions possibly called when a function of this type is called). A type environment T is a finite map from identifiers to types.

$$\begin{array}{lll l}
c \in & \mathit{Control} & = \emptyset \mid \{\mathrm{n}\} \mid cv \mid c \cup c' & \mathit{Control} \\
t \in & \mathit{Type} & = \mathit{unit} \mid tv \mid ref(t) \mid t \xrightarrow{c} t' & \mathit{Type} \\
T \in & \mathit{Tenv} & = \mathit{Id} \rightarrow \mathit{Type} & \mathit{Type\ environment}
\end{array}$$

Given a type environment T, the inference rules of the static semantics associate the expression `e` with its type and control-flow information:

$$T \vdash \mathtt{e} : t, c$$

The crucial rules are the *(abs)* and *(app)* rules for lambda abstraction and application. In the abstraction case, the current function name is added to the functions called by the lambda body; the resulting set is the latent control-flow effect of the lambda expression. When such a function is applied, in the *(app)* rule, this latent control-flow information is used to determine the functions possibly called while evaluating the function body.

Types and control-flow effects of variables bound to side-effect free expressions in `let` forms can be polymorphic. A simple policy, based on the expansiveness of expressions [14], is chosen to detect if expressions have side-effects or not. A *non-expansive* expression is syntactically guaranteed not to allocate references. Variables and lambda-abstractions are non-expansive expressions. By extension, a `let` expression is non-expansive if and only if both of its binding expression and its body expression are non-expansive.

Non-expansive `let` expressions, which can be generalized over, are handled by syntactic substitution of the binding for the variable in the body. This avoids the complication of introducing sophisticated type schemes inside the static semantics. This simple technique provides an equivalent way of expressing the polymorphic types of non-expansive expressions bound in `let` bindings. We write `e'[e/x]` for the textual substitution of `e` for `x` in `e'`.

Subeffecting is introduced by the *(does)* rule which can be used whenever a type or effect mismatch occurs in the application rule *(app)* and the assignment rule *(set)*. This allows control-flow information to be augmented when needed.

$$(var) : \frac{\mathtt{x} \in Dom(T)}{T \vdash \mathtt{x} : T(\mathtt{x}), \emptyset}$$

$$(abs) : \frac{T_{\mathtt{x}}\{\mathtt{x} \mapsto t'\} \vdash \mathtt{e} : t, c}{T \vdash (\mathtt{lambda\ n\ (x)\ e}) : t' \overset{\{\mathtt{n}\} \cup c}{\rightarrow} t, \emptyset}$$

$$(app) : \frac{\begin{array}{l} T \vdash \mathtt{e} : t' \overset{c''}{\rightarrow} t, c \\ T \vdash \mathtt{e}' : t', c' \end{array}}{T \vdash (\mathtt{e\ e}') : t, c \cup c' \cup c''}$$

$$(let) : \frac{\begin{array}{l} \neg expansive[\![\mathtt{e}]\!] \\ T \vdash \mathtt{e} : t, c \\ T \vdash \mathtt{e}'[\mathtt{e}/\mathtt{x}] : t', c' \end{array}}{T \vdash (\mathtt{let\ (x\ e)\ e}') : t', c'}$$

$$(ilet) : \frac{\begin{array}{l} expansive[\![\mathtt{e}]\!] \\ T \vdash \mathtt{e} : t, c \\ T_{\mathtt{x}} \cup \{\mathtt{x} \mapsto t\} \vdash \mathtt{e}' : t', c' \end{array}}{T \vdash (\mathtt{let\ (x\ e)\ e}') : t', c \cup c'}$$

$$(does) : \frac{T \vdash \mathtt{e} : t, c}{T \vdash \mathtt{e} : t, c \cup c'}$$

$$(new) : \frac{T \vdash \mathtt{e} : t, c}{T \vdash (\mathtt{new\ e}) : ref(t), c}$$

$$(get) : \frac{T \vdash \mathtt{e} : ref(t), c}{T \vdash (\mathtt{get\ e}) : t, c}$$

$$(set) : \frac{\begin{array}{l} T \vdash \mathtt{e} : ref(t'), c \\ T \vdash \mathtt{e}' : t', c' \end{array}}{T \vdash (\mathtt{set\ e\ e}') : unit, c \cup c'}$$

The consistency of the dynamic and static semantics states that, if a function name n occurs in the control-flow effect of an expression, then the function n may be called while evaluating the expression. The proof is similar to [13].

5 Escape Analysis

The compile-time knowledge of possible control-flow behaviors permits escape analysis to be performed, even in the presence of imperative constructs and higher-order functions.

5.1 Identifying Escaping Functions

A function *escapes* if its value is accessible outside the lexical scope of any of its free variables. We use the control-flow information inferred by our static semantics to determine an approximation of this property. More precisely, we identify all escaping functions and their escape-level and escape-set. The *escape-level* of an escaping function is the smallest lexical level at which the function escapes (or *infinity* if the function does not escape) while its *escape-set* is the set of the free variables of the function body that outlive their scope.

To compute this information, we use two environments, *LE* and *EE*. The *lexical environment LE* maps identifiers to the integer lexical levels at which they are bound. The *escape-environment EE* maps a lexical level *lev* defining a lambda expression to the function type $t \xrightarrow{\emptyset} t'$; this type records, via the latent control-flow effects possibly present in t or t', the names of all of the functions that may escape at the level *lev*. A function n, defined at level *lev'*, escapes at level *lev* by being either part of the value returned at level *lev* (its name is free in t') or stored in a location bound at lexical level *lev* (its name is free in t); the function n is then said to escape from *lev'* to *lev*.

Given a lexical environment *LE* and an escape-environment *EE*, the algorithm $\mathcal{I}$ updates, for an expression e at level *lev*, the identification function i. This *identification function* maps a function name to the pair of its escape-level and escape-set. Otherwise, a function can escape to multiple lexical levels; conservatively, the escape-level is the minimum of them. The escape-set collects all of the free variables of a function bound at a lexical level larger than the escape-level. These two escape attributes are conservative approximations of their dynamic counterparts.

We assume in $\mathcal{I}$ that the expression is completely typed, the type and control-flow information having been previously inferred by the control-flow effect system, and that expansive `lets` are desugared into lambda applications while non-expansive ones are explicitly substituted.

```
$\mathcal{I}$(e) $LE$ $EE$ $lev$ $i$ =
    case e in
    x $\Rightarrow$ $i$
    (e e') $\Rightarrow$ $\mathcal{I}$(e) $LE$ $EE$ $lev$  ($\mathcal{I}$(e') $LE$ $EE$ $lev$ $i$)
    (lambda n (x:t) e':t') $\Rightarrow$
```

```
let $i' = \mathcal{I}(\mathtt{e}')\ (LE\{\mathtt{x}\mapsto lev\})\ (EE\{lev\mapsto \mathtt{t}\xrightarrow{\emptyset}\mathtt{t'}\})\ (lev+1)\ i$
let el = $\mathtt{Min}\{lev \mid \mathtt{n} \in \mathit{fn}(EE(lev))\}$
let es = $\{\mathtt{y} \in \mathit{fv}(\mathtt{e}) \mid LE(\mathtt{y}) \geq \mathtt{el}\}$
$i'\{\mathtt{n}\mapsto(\mathtt{el},\ \mathtt{es})\}$
```

where *fv* computes the set of free variables of expressions and environments, while *fn* restricts this set to function names. By convention, `Min{}` is defined to be *infinity*. The identification function of a whole program expression p is given by calling $\mathcal{I}$ on p with empty environments, lexical level 0 and a bottom identification function.

5.2 Allocation Strategy

The previously computed escape attributes, escape-level and escape-set, can be used to efficiently allocate closure environments. First, the environment of a non-escaping function can be stack-allocated. Second, for an escaping function, only the bindings of the variables that appear in its escape-set need to be heap-allocated in the closure environment; the others can, as before, be stack-allocated. These two code and space optimizations can be extremely worthwhile, especially in large programs that need separate compilation.

6 Conclusion

We introduced a new static system that reconstructs the types and control-flow information of expressions in an implicitly typed functional language with imperative operations. Using the type and effect framework, this analysis can be performed in the presence of separate compilation and higher-order functions, where other methods fail. We showed how generic polymorphism in `let` bindings can be introduced when computing control-flow information. Although not presented here, recursive functions can be dealt with easily. As an application, we described how closure environments can be efficiently stack-allocated using control-flow information.

We are currently implementing this algorithm in a full-fledged compiler, and working to extend our method to deal with control-flow polymorphism for lambda-bound functions,

Acknowledgments

We thank Jean-Pierre Talpin for his advice and Vincent Dornic, Susan Flynn-Hummel, Mark A. Sheldon and Jim O'Toole for their remarks on drafts of this paper.

References

[1] Appel, A. W. *Compiling with Continuations.* Princeton University, 1992.

[2] Baker, H.G. "CONS Should not CONS its Arguments, or, a Lazy Alloc is Smart Alloc". In *ACM SIGPLAN Notices, Volume 27, No.3* ,March, 1992

[3] Dornic, V., and Jouvelot, P. "Polymorphic Time Systems for Estimating Program Complexity". In *JTASPEFL'91, Bordeaux, France,* October 1991.

[4] Gifford, D. K., Jouvelot, P., Lucassen, J. M., and Sheldon, M. A. "FX-87 Reference Manual". In *MIT/LCS/TR-407,* MIT Laboratory for Computer Science, September 1987.

[5] Jouvelot, P., and Gifford, D. K. "Reasoning about Continuations with Control Effects". In *Proceedings of the 1989 ACM SIGPLAN Int. Conf. on Prog. Lang. Desi. and Impl.*. ACM, New-York, 1989.

[6] Jouvelot, P., and Gifford, D. K. "Algebraic reconstruction of types and effects". In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages.* ACM, New-York, 1991.

[7] Kranz, A. D. *ORBIT: An Optimizing Compiler for Scheme.* PhD thesis, Yale University. February 1988.

[8] Lucassen, J. M., and Gifford, D. K. "Polymorphic Effect Systems". In *Proceedings of the 1988 ACM Conference on Principles of Programming Languages.* ACM, New-York, 1988.

[9] Plotkin, G. "A structural approach to operational semantics". In *Technical report DAIMI-FN-19.* Aarhus University, 1981.

[10] Rees, J., and Clinger W., Editors. *Fourth Report on the Algorithmic Language Scheme.* September 1988.

[11] Shivers, O. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, University of Yale. May 1991.

[12] Steele, G. "Rabbit: A Compiler for Scheme". In *MIT-AI Technical Report No. 474.* MIT Laboratory for Computer Science, May 1978.

[13] Talpin, J. P., and Jouvelot, P. "Polymorphic Type, Region and Effect Inference". To appear in the *Journal of Functional Programming,* Cambridge University Press, 1992. Also in *Technical Report EMP-CRI-E150,* Ecole Nationale Supérieure des Mines de Paris, February 1991 (revised December 1991).

[14] Tofte, M. *Operational semantics and polymorphic type inference.* PhD Thesis, University of Edinburgh, 1987.