A Lifetime Analysis for Higher-Order Languages

(Extended Abstract)

John Hannan and Patrick Hicks
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802 USA
and
David Liben-Nowell
Department of Computer Science
Cornell University
Ithaca, NY 14853 USA

1. INTRODUCTION

The rich structure of higher-order functional languages complicates the construction of efficient implementations for them. In particular, the simple memory management policy of block-structured languages, which allocates storage for variables upon block entry and then deallocates it upon block exit, cannot be applied blindly. Alternative techniques, usually including garbage collection, must be applied. But even with the use of garbage collection, care must be taken in the representations of run-time data structures to support efficient use of space. Lifetime information for variables plays a crucial role in determining these representations.

The feature of higher-order functional languages which contributes the greatest complication of storage management is the support of functions (with free variables) as first-class values. A traditional implementation technique to support these builds function closures containing both code and information on how to access the values of non-local variables in the function. Closure conversion is the process of transforming a program containing functions with free variables into one containing closures. An important decision during closure conversion is the representation of closures. Recent work has considered type systems for closure conversion [Minamide et al. 1996], and other work has demonstrated that the choice of closure representation can have a dramatic impact in the space required for closures during the execution of a program [Shao and Appel 1994]. Some approaches to closure conversion use lifetime information of variables to determine the appropriate representation of closures and other run-time data structures. By using lifetime information the closure conversion process can choose closure representations which are space efficient, meaning that parts of closures can be deallocated when they are no longer needed.

The goal of our work is to develop a type system which, along with providing the usual type information, provides lifetime information for variables. In addition, we wish to demonstrate the correctness of this lifetime information, and, eventually, the correctness of various space-efficient closure representation strategies. We do this by showing a type consistency result and the correctness of an operational semantics that "throws away" parts of closures, which, according to the lifetime

This work has been supported by NSF CAREER Grant #CCR-9502356. This paper has been submitted for publication. Comments are welcome. A full version of this paper is available.

analysis, are no longer needed. This operational semantics is not practical, but it does serve as a setting to verify the lifetime information.

This work generalizes previous work which considered only whether a formal parameter could escape its own scope [Hannan 1995; 1997]. We proved the correctness of that analysis by demonstrating the equivalence of a standard operational semantics and an operational semantics which exploited the information provided by the analysis. The present lifetime analysis generalizes this previous result in two ways. First, we provide more precise information about the potential use of variables by indicating the potential lifetime for each variable. This information has greater potential for application (as described above) than the escape analysis. Second, we generalize the correctness statement, making it independent of an operational semantics exploiting the lifetime information. The uniform nature in which we treat all variables here simplifies the proof of correctness (compared to [Hannan 1997]).

To give a simple characterization of lifetime as a static property we associate with each function in a program a positive number called the function's *static depth*. This is essentially its de Bruijn index, i.e., the number of lexically enclosing function definitions plus 1. Then for each parameter of a function we determine its *lifetime depth* which is a natural number corresponding to the smallest static depth in the program at which this variable could possibly be live. (A lifetime of 0 indicates that the parameter can be live in a closure returned as the result of the entire program.) The effect of our analysis is to annotate a program with lifetime depths for each function parameter. Admittedly, this notion of lifetime is simplistic, and a refinement would be to provide finer distinction between lifetimes at a given static depth.

To determine lifetime information we use annotated types. An extension of simple types, these types include information about live variables. For a given type ϕ we can compute the set of live variables of the type, denoted $\mathsf{LV}(\phi)$. A type consistency result then provides information about the live variables in a value computed at a given static depth. Informally, if an expression e occurring at static depth n has type ϕ, and variable x is in $\mathsf{LV}(\phi)$, then x can be live in a value produced at this depth. Hence its lifetime is at most n. (Recall that a smaller lifetime depth corresponds to a longer lifetime, i.e., a lifetime that extends to a larger scope in the program.) More interesting is that if x is not in $\mathsf{LV}(\phi)$ then x cannot be live in a value produced at this depth. By organizing type information appropriately we can strengthen this conclusion to infer that x cannot be live at this depth or any lesser depth. Thus our analysis simply needs to find the largest enclosing expression whose type does not contain x live to determine the lifetime depth of x.

2. EXPRESSIONS, TYPES, AND OPERATIONAL SEMANTICS

We consider a simple higher-order functional language with call-by-value parameter passing.

$$e \ ::= \ c \mid x \mid \mathsf{if}\ e\ e\ e \mid \lambda x.e \mid e \mathbin{@} e \mid \mu f.\lambda x.e$$

where c ranges over some set of constants and $(e_1 \mathbin{@} e_2)$ represents the application of e_1 to e_2. The fixed-point operator μ provides for recursive function definitions. Types are implicit, à la Curry, and terms can be typed via a standard typing system using judgments of the form $\Gamma \rhd e : \tau$.

We introduce an annotated form of this language:

$$m \ ::= \ c \mid x^k \mid \mathsf{if}\ m\ m\ m \mid \lambda^d x^k.m \mid m \mathbin{@} m \mid \mu f^{k'}.\lambda^d x^k.m$$

The superscripts k and d range over natural and positive numbers, respectively. We say that these numbers annotate variables, λ-abstractions, and fixed-points. These annotations make lifetime information explicit as follows. A λ-abstraction contains two annotations: the static nesting depth d of this function and the lifetime k of the bound variable (formal parameter) x. The lifetime

$$\frac{}{\rho \rhd_n c \hookrightarrow c} \qquad \frac{\rho(x^k) = v}{\rho \rhd_n x^k \hookrightarrow v}$$

$$\frac{\rho \rhd_n m_1 \hookrightarrow \mathsf{true} \quad \rho \rhd_n m_2 \hookrightarrow v}{\rho \rhd_n \mathsf{if}\ m_1\ m_2\ m_3 \hookrightarrow v} \qquad \frac{\rho \rhd_n m_1 \hookrightarrow \mathsf{false} \quad \rho \rhd_n m_3 \hookrightarrow v}{\rho \rhd_n \mathsf{if}\ m_1\ m_2\ m_3 \hookrightarrow v}$$

$$\frac{}{\rho \rhd_n \lambda^n x^k.m \hookrightarrow \left[\rho, \lambda^n x^k.m\right]}$$

$$\frac{}{\rho \rhd_n \mu f^{k_1}.\lambda^{n+1} x^{k_2}.m \hookrightarrow \left[\rho, \mu f^{k_1}.\lambda^{n+1} x^{k_2}.m\right]}$$

$$\frac{\rho \rhd_n m_1 \hookrightarrow \left[\rho', \lambda^d x^k.m'\right] \qquad \rho \rhd_n m_2 \hookrightarrow v_2 \qquad \rho'\{x^k \mapsto v_2\} \rhd_{d+1} m' \hookrightarrow v}{\rho \rhd_n m_1 \,@\, m_2 \hookrightarrow v}$$

$$\frac{\begin{array}{c}\rho \rhd_n m_1 \hookrightarrow \left[\rho', \mu f^{k_1}.\lambda^d x^{k_2}.m'\right] \qquad \rho \rhd_n m_2 \hookrightarrow v_2 \\ \rho'\{f^{k_1} \mapsto \left[\rho', \mu f^{k_1}.\lambda^d x^{k_2}.m'\right]\}\{x^{k_2} \mapsto v_2\} \rhd_{d+1} m' \hookrightarrow v\end{array}}{\rho \rhd_n m_1 \,@\, m_2 \hookrightarrow v}$$

Fig. 1. Operational Semantics

annotation k indicates the shallowest static nesting depth in which this variable can be accessed (through some escaping function closure). Variables bound by the μ operator also contain a lifetime annotation which can be interpreted in the same manner.

We use a traditional operational semantics for the language, defining the judgment $\rho \rhd_n m \hookrightarrow v$ in which v is either a constant or a function closure $\left[\rho, \lambda^d x^k.m\right]$. The subscript n simply denotes the length of environment ρ plus 1, or, equivalently, the static nesting depth of the term m plus 1. The axiomatization of this judgment as a deductive system is standard, except for the presence of annotations, and is given in Fig. 1. For this deductive system, and others introduced subsequently, we write $\Pi :: \rho \rhd_n m \hookrightarrow v$ to indicate that Π is a deduction of the judgment $\rho \rhd_n m \hookrightarrow v$.

The operational semantics maintains the static depth of the current expression being evaluated by using the static depths which annotate λ-abstractions. The application rules clearly demonstrate this. We use this static depth information to provide a reference point from which to reason about which variables' lifetimes are still in effect. If we have a deduction of $\rho \rhd_n m \hookrightarrow v$ then any variables with lifetime $k > n$ which occur in v (in closures) can be eliminated from this value, as these variables cannot be needed at this depth. This holds true even if the value is subsequently passed down to a depth $\geq k$, since, if this were the case, then these variables would have been given a lifetime of n indicating that their presence is required at depth n, even if only to be passed down to a greater depth.

3. THE LIFETIME ANALYSIS

The goal of the analysis is to annotate terms with accurate lifetime information. We use a type system which takes an unannotated term and translates it into an annotated form. We compute appropriate lifetime annotations by using annotated type information. Let Δ range over sets of annotated variables. The set of annotated types is defined as

$$\phi \ ::= \ \iota \mid \phi \xrightarrow{\Delta} \phi$$

$$\frac{(c : |\phi|) \in \Sigma \qquad \mathsf{max}(\mathsf{LV}(\phi)) \le n}{\Gamma \rhd_n c : (\phi, \{\}) \Rightarrow c} \qquad \frac{\Gamma(x^k) = \phi}{\Gamma \rhd_n x : (\phi, \{x^k\}) \Rightarrow x^k}$$

$$\frac{\Gamma \rhd_n e_1 : (\mathsf{bool}, \Delta_1) \Rightarrow m_1 \quad \Gamma \rhd_n e_2 : (\phi, \Delta_2) \Rightarrow m_2 \qquad \Gamma \rhd_n e_3 : (\phi, \Delta_3) \Rightarrow m_3}{\Gamma \rhd_n \mathsf{if}\ e_1\ e_2\ e_3 : (\phi, \Delta_1 \cup \Delta_2 \cup \Delta_3) \Rightarrow \mathsf{if}\ m_1\ m_2\ m_3}$$

$$\frac{\Gamma\{x^k : \phi_1\} \rhd_{n+1} e : (\phi_2, \Delta) \Rightarrow m \qquad \begin{array}{l} \Delta \subseteq \Delta' \quad \mathsf{max}(\Delta') \le n \\ \mathsf{max}(\mathsf{LV}(\phi_1)) \le k \le n \\ \mathsf{max}(\mathsf{LV}(\phi_2)) \le n \end{array}}{\Gamma \rhd_n \lambda x.e : (\phi_1 \xrightarrow{\Delta'} \phi_2, \{\}) \Rightarrow \lambda^n x^k.m}$$

$$\frac{\Gamma\{f^k : \phi\} \rhd_{n+1} e : (\phi, \{\}) \Rightarrow m \qquad \mathsf{max}(\mathsf{LV}(\phi)) \le k \le n}{\Gamma \rhd_n \mu f.e : (\phi, \{\}) \Rightarrow \mu^n f^k.m}$$

$$\frac{\Gamma \rhd_n e_1 : (\phi_1 \xrightarrow{\Delta} \phi_2, \Delta_1) \Rightarrow m_1 \qquad \Gamma \rhd_n e_2 : (\phi_1, \Delta_2) \Rightarrow m_2}{\Gamma \rhd_n (e_1 @ e_2) : (\phi_2, \Delta \cup \Delta_1 \cup \Delta_2) \Rightarrow m_1 @ m_2}$$

Fig. 2. The Lifetime Analysis

in which ι ranges over some base types. We let $\mathsf{LV}(\phi)$ denote the set of *live* variables occurring in the type ϕ, defined as follows:

$$\mathsf{LV}(\iota) = \{\} \qquad \mathsf{LV}(\phi_1 \xrightarrow{\Delta} \phi_2) = \Delta \cup \mathsf{LV}(\phi_2).$$

Note that the set of live variables of a function type $\phi_1 \xrightarrow{\Delta} \phi_2$ does not contain the live variables of ϕ_1. Intuitively, a value of type $\phi_1 \xrightarrow{\Delta} \phi_2$ is a function closure which when called requires at most the variables in Δ and returns a value of type ϕ_2. Live variables of ϕ_1 may or may not appear in Δ and $\mathsf{LV}(\phi_2)$, depending on how the argument to the function is used. However the function is used, any variables required by the argument during the function call will appear in Δ.

We extend the notion of live variables to type contexts. If Γ is a type context mapping annotated variables to annotated types, then the live variables of the context, written $\mathsf{LV}(\Gamma)$, is the set $\bigcup(\mathsf{LV}(\Gamma(x^k)) \cup \{x^k\} \mid x^k \in \mathsf{dom}(\Gamma))$. This consists of the domain of Γ and the free variables in the types in the range of Γ.

We define a static analysis as a type system which takes a raw λ-term and infers an annotated type and lifetime annotations for the term. The analysis uses a judgment of the form $\Gamma \rhd_n e : (\phi, \Delta) \Rightarrow m$ in which Γ is a type context mapping annotated variables to annotated types, e is a (raw) λ-term, ϕ is an annotated type, Δ is a set of annotated variables, and m is the annotated form of e, containing lifetime annotations. The judgment can be understood as stating: "Under the type assumptions of Γ, expression e has type ϕ, accesses at most the variables in Δ (during call-by-value evaluation to weak head normal form) and translates to term m."

We axiomatize this judgments via the deductive systems of Figure 2. We assume a signature Σ which maps constants to unannotated types. The essential task of the deductive system, beyond performing traditional type inference, is to generate constraints for the set Δ and the variables occurring in the type ϕ in a judgment. Informally, if we can derive $\Gamma \rhd_n e : (\phi, \Delta) \Rightarrow m$ then the rules ensure the following constraints:

(1) the evaluation of e requires at most the variables in Δ; and

(2) the lifetime depths of variables in ϕ are at most n.

That the latter property holds is not immediately obvious, and in the next section we prove that it holds under certain conditions.

Most of the rules are self-explanatory, keeping in mind the two items above. In the rule for variables, we use the familiar notation $\Gamma(x^k) = \phi$ to indicate that in Γ, x is annotated with lifetime depth k and is mapped to ϕ. As an alternative, we could let Γ map unannotated variables x to a pair (k, ϕ). This structure more accurately reflects the use of Γ. In the rule for application, if we can derive $\Gamma \rhd_n : e_1(\phi_1 \xrightarrow{\Delta} \phi_2, \Delta_1) \Rightarrow m_1$ and $\Gamma \rhd_n e_2 : (\phi_1, \Delta_2) \Rightarrow m_2$ then we can conclude $\Gamma \rhd_n (e_1 \mathbin{@} e_2) : (\phi_2, \Delta \cup \Delta_1 \cup \Delta_2) \Rightarrow (m_1 \mathbin{@} m_2)$ because:

—the evaluation of $(e_1 \mathbin{@} e_2)$ requires the evaluation of e_1 (which requires Δ_1) to a function;

—the evaluation of e_2 (which requires Δ_2); and

—the evaluation of the resulting function call (which requires Δ).

The rule for λ-abstraction specifies constraints for the lifetime information of variables. This rule includes the side conditions $\Delta \subseteq \Delta'$, $\mathsf{max}(\Delta') \leq n$, $\mathsf{max}(\mathsf{LV}(\phi_1)) \leq k \leq n$, and $\mathsf{max}(\mathsf{LV}(\phi_2)) \leq n$. The first two conditions allow the set Δ to be weakened to a larger set, but restricts this larger set to contain only variables whose lifetime depths are bounded by n. We require this weakening to support the typing of all expressions which can be typed via the traditional simple-type system. The last two conditions provide sanity checks on types. The condition $\mathsf{max}(\mathsf{LV}(\phi_1)) \leq k$ ensures that if x has lifetime k, then any live variables in a value bound to x have lifetime at least as long as ($\leq$) this. The condition $k \leq n$ ensures that the variable's lifetime is at least as long as its static depth, since it certainly can be no shorter. The final condition $\mathsf{max}(\mathsf{LV}(\phi_2)) \leq n$ ensures that any value produced as a result of this function must have a lifetime at least as long as the static depth of the function.

An example of a derivable judgment more clearly demonstrates the information generated by the system. The following example demonstrates a λ-term which contains an argument's lifetime depth is equal to neither 0 nor its function's static depth (i.e. an argument that escapes from its scope but not all the way to the top-level).

$$\bullet \rhd_1 \lambda x.((\lambda f.f \mathbin{@} x) \mathbin{@} ((\lambda y.\lambda z.y) \mathbin{@} (\lambda g.x))) : (\alpha \xrightarrow{\{f,x,y\}} (\beta \xrightarrow{\{x\}} \alpha), \{\}) \Rightarrow \\ \lambda^1 x^0.((\lambda^2 f^2.f \mathbin{@} x) \mathbin{@} ((\lambda^2 y^1.\lambda^3 z^3.y) \mathbin{@} (\lambda^2 g^2.x)))$$

The variable y has a lifetime which extends to the enclosing function definition because the value of the expression $((\lambda^2 y^1.\lambda^3 z^3.y) \mathbin{@} (\lambda^2 g^2.x))$ is a function closure containing y. This closure is used at static depth 1 (but not at static depth 0). The variable x can be live in a closure (for the function $(\lambda^2 g^2.x)$) returned as the value of the entire expression, so its lifetime is 0.

We have developed an algorithm for this analysis. Details of it can be found in [Hannan et al. 1997].

4. CORRECTNESS

To demonstrate the correctness of the analysis we prove that the analysis is safe, i.e., if a variable x is assigned a lifetime depth k, then x is not live at a depth $n < k$. We show this safety property via a type consistency result. The structure of the correctness proof is similar to the one for our escape analysis presented in [Hannan 1997]. This similarity is not surprising, as the lifetime analysis is a generalization of the escape analysis.

To state our correctness property we first must define three auxiliary definitions. The first regards the uniqueness of free and bound variables occurring in terms. The second characterizes a useful property of the type contexts we manipulate. The third defines the consistency of values and annotated types. We also introduce the notation $\mathsf{max}(S)$ (for a set S of annotated variables) to denote the maximum integer annotating a variable in S. Also, let $\mathsf{Rectify}(e)$ be true of a term e when all the bound variables in e are unique and distinct from all the free variables of e.

Definition 4.1 (Distinct). The relation Distinct is the smallest relation satisfying

(1) $\mathsf{Distinct}(\bullet)$ for empty environments and empty contexts;
(2) $\mathsf{Distinct}(\rho\{x^k \mapsto v\})$ if $\mathsf{Distinct}(\rho)$, $\mathsf{Distinct}(v)$, and $x \notin \mathsf{dom}(\rho)$;
(3) $\mathsf{Distinct}(\Gamma\{x^k : \phi\})$ if $\mathsf{Distinct}(\Gamma)$ and $x \notin \mathsf{dom}(\Gamma)$;
(4) $\mathsf{Distinct}(c)$ for all constants c;
(5) $\mathsf{Distinct}([\rho, e])$ if $\mathsf{Distinct}(\rho)$, $\mathsf{Rectify}(e)$, and $\mathsf{BV}(e) \cap \mathsf{dom}(\rho) = \{\}$.

In cases (2) and (3) we specify $x \notin \mathsf{dom}(\rho)$ and $x \notin \mathsf{dom}(\Gamma)$ (and not x^k) to indicate that no occurrence of x (with any label) occurs in these domains. We also may write $x \notin \Delta$ to specify that no labeling of x occurs in set Δ. We require distinct objects to avoid any confusion between occurrences of live variables in a type and the λ-bindings to which they correspond.

Definition 4.2 (Proper Contexts).

(1) $\mathsf{Proper}(\bullet, 0)$;
(2) $\mathsf{Proper}(\Gamma\{x^k : \phi\}, n)$ if $k \leq n$, $\mathsf{max}(\mathsf{LV}(\phi)) \leq k$, and $\mathsf{Proper}(\Gamma, n-1)$.

Intuitively, the natural number n is the static depth at which the variable x occurred in a term. A proper context guarantees that the type of a variable declared at static depth n has only live variables with lifetimes bounded by n.

To define the consistency of a value and a type, and also the consistency of an environment and a type context, we relate them with respect to a set Δ of annotated variables and a static depth n.

Definition 4.3 (Consistency). The relations $\mathsf{Consistent}(v, \phi, \Delta, n)$ and $\mathsf{Consistent}(\rho, \Gamma, \Delta, n)$ are the smallest relations satisfying:

(1) $\mathsf{Consistent}(c, \phi, \Delta, n)$ if $(c : |\phi|) \in \Sigma$;
(2) (a) $\mathsf{Consistent}(\left[\rho, \lambda^d x^k.m\right], \phi, \Delta, n)$ if there exist $\lambda x.e$ and Γ such that $\mathsf{Proper}(\Gamma, d-1)$, $\Xi{::}\Gamma \rhd_d \lambda x.e : (\phi, \{\}) \Rightarrow \lambda^d x^k.m$, and $\mathsf{Consistent}(\rho, \Gamma, \Delta \cap \mathsf{LV}(\phi), n)$;
   (b) $\mathsf{Consistent}(\left[\rho, \mu f^{k_1}.\lambda^d x^{k_2}.m\right], \phi, \Delta, n)$ if there exist $\mu f.\lambda x.e$ and Γ such that $\mathsf{Proper}(\Gamma, d-2)$, $\Xi{::}\Gamma \rhd_{d-1} \mu f.\lambda x.e : (\phi, \{\}) \Rightarrow \mu f^{k_1}.\lambda^d x^{k_2}.m$, and $\mathsf{Consistent}(\rho, \Gamma, \Delta \cap \mathsf{LV}(\phi), n)$;

(3) $\mathsf{Consistent}(\bullet, \bullet, \Delta, n)$;
(4) $\mathsf{Consistent}(\rho\{x^k \mapsto v\}, \Gamma\{x^k : \phi\}, \Delta, n)$ if $x^k \in \Delta$, $k \leq n$, $\mathsf{max}(\mathsf{LV}(\phi)) \leq k$, $\mathsf{Consistent}(v, \phi, \Delta, n)$, and $\mathsf{Consistent}(\rho, \Gamma, \Delta, n)$.
(5) $\mathsf{Consistent}(\rho\{x^k \mapsto v\}, \Gamma\{x^k : \phi\}, \Delta, n)$ if $x \notin \Delta$ and $\mathsf{Consistent}(\rho, \Gamma, \Delta, n)$.

This definition generalizes the traditional concept of type consistency for values by incorporating requirements that lifetime annotations for live variables (those in Δ) be consistent with (less than or equal to) static depth n. Note that this definition is well-founded and so we can reason by induction over it.

To demonstrate the correctness of our analysis we show that if an expression e at static depth n has type ϕ and e evaluates to v then v is consistent with type ϕ at depth n. In other words, the live variables of v are consistently annotated with lifetime information for this value to appear at depth n.

THEOREM 4.4 (TYPE CONSISTENCY). *If* $\mathsf{Proper}(\Gamma, n-1)$, $\Gamma \rhd_n e : (\phi, \Delta) \Rightarrow m$, $\Delta \subseteq \Delta_0$, $\mathsf{Consistent}(\rho, \Gamma, \Delta_0, n-1)$, $\mathsf{Distinct}([\rho, m])$, *and* $\rho \rhd_n m \hookrightarrow v$ *then* $\mathsf{Consistent}(v, \phi, \Delta_0, n)$.

The complete proof can be found in [Hannan et al. 1997].

REFERENCES

HANNAN, J. 1995. A type-based analysis for stack allocation in functional languages. In *Proceedings of the 2nd International Static Analysis Symposium*. Springer-Verlag LNCS 983, 172–188.

HANNAN, J. 1997. A type-based escape analysis for functional languages. *Journal of Functional Programming*. To appear.

HANNAN, J., HICKS, P., AND LIBEN-NOWELL, D. 1997. A lifetime analysis for higher-order languages. Submitted.

MINAMIDE, Y., MORRISETT, G., AND HARPER, R. 1996. Typed closure conversion. In *23rd ACM Symposium on Principles of Programming Languages*. St. Petersburg, FL, 271–283.

SHAO, Z. AND APPEL, A. W. 1994. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*. ACM, ACM Press, 150–161.