Typed stack allocation

Brian Brooks

University of Colorado Boulder

Follow this and additional works at: https://scholar.colorado.edu/csci_ugrad

Recommended Citation

Brooks, Brian, "Typed stack allocation" (2011). Computer Science Undergraduate Contributions. 36.
https://scholar.colorado.edu/csci_ugrad/36

This Thesis is brought to you for free and open access by Computer Science at CU Scholar. It has been accepted for inclusion in Computer Science Undergraduate Contributions by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.
1 Introduction

The verification of memory safety—no program will crash when referencing memory—falls into two categories. A runtime system may enforce memory safety at a cost. Or, we may rely on error-prone programmers.

Runtime overhead, most noticeably garbage collection, is undesirable in embedded or real-time systems. The frequency of collection points is not deterministic. Languages that rely on garbage collection suffer from poor cache locality, occupy more memory than necessary, and usually spend more time copying data throughout memory hierarchy than performing meaningful computations.

With enough memory, Appel [2] claims that it is more expensive to explicitly free memory than to leave it for a garbage collector. While garbage collection yields simpler and cleaner syntax, it is worth understanding the circumstances under which garbage collection can be 'cheaper' than manually deallocating memory. The result of Appel's work states that garbage collection becomes less expensive when you have seven times as much memory as you have data within your program.

Relying on programmers to safely and explicitly allocate, access, and deallocate memory has been the infamous source of unreliable software. There are many advantages to explicit memory management e.g. deallocating memory immediately after it’s last access instead of waiting for a GC to reclaim it. However, this ad-hoc method of verification has proven over time that secure software systems require a machine to verify memory safety. Why is it that the most popular techniques for guaranteeing memory safety impose runtime overhead?

applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)

applyTwice takes two arguments: \( f :: (a \rightarrow a) \) and \( x :: a \). When we apply applyTwice to a function \( f \), there is no guarantee that \( f \) takes exactly one parameter. We may end up with a partially applied function if \( f \) takes two or more parameters. Because \( f \) is unknown during compile-time we may allocate \( f \) in the heap. Higher-order languages tend to allocate most values in the heap.

There has yet to be a silver-bullet method for statically determining optimal lifetimes of values. Thus, languages rely on garbage collection to periodically manage memory during program execution.

1.1 Contributions

Our approach towards memory safety “for free” only permits values that may be allocated on the runtime stack. We do not rely on a garbage collector or runtime checks to guarantee memory safety.

We define an abstract machine that can model the runtime stack during execution of programs. We then provide a type and effect system for describing memory-safe programs with respect to the abstract machine.
2 Modeling stack allocation

In imperative languages such as C and C++, the lifetimes of local variables are controlled by the basic block that they are defined in. In the function,

```c
int f() {
    int a = 0;
    return 42;
}
```
the variable `a` will be allocated on the stack within `f()`'s activation record. Once the body of `f()` is done evaluating, we return the value 42 and the activation record as well as `a` are popped off the stack. Since, we can statically determine `a`'s lifetime, we may infer points of allocation and deallocation. As a property of memory safety, we must ensure that we do not access `a` before it has been allocated and after it has been deallocated.

Compilers for type-unsafe languages allow programmers to write code that returns the address of a local variable. GCC 4.0.1 does not give a warning or error for the following function,

```c
int *x() {
    int a;
    int *b = &a;
    return b;
}
```

In the simply typed lambda calculus, the syntactic basic block is lambda abstraction: \( \lambda x . e \) where \( x \) is input and \( e \) is output. Due to the operational semantics of the simply typed lambda calculus, \( x \) may live longer than the evaluation of the lambda abstraction that allocated its storage. Consider evaluation of this term:

\[
\lambda x . \lambda y . x 4 8
\]

\( x \) will be bound to 4, \( y \) will be bound to 8, and we need to evaluate \( x \)–which must exist in memory somewhere. Since the two lambda abstractions have already been pushed onto and popped off of the runtime stack—in LIFO order—\( x \) must be allocated in the heap.

The behavior of a runtime stack is that parameter values are allocated upon function application and are deallocated when the function returns. Since we are only interested in stack allocation, we must be able to describe not only function application but also returning from functions in our operational semantics.

2.1 ECD machine

Consider the syntax of the call-by-value \( \lambda \)-calculus with integers. Expressions are either integers, variables, abstractions, or applications:

\[
i \in \text{Integers}\\
x \in \text{Variables}\\
ee \in \text{Expressions} ::= i \mid x \mid \lambda x . e \mid e \ e
\]

Expressions decompose into a context \((C)\) and a redex \((r)\). A context is simply an expression with a hole \((\_\_\_\_\_\_\_\_)\) which serves as a placeholder for a redex, i.e. \( e = C[r] \). The following grammar defines contexts that preserve left-to-right evaluation of our calculus:

\[
C \in \text{Contexts} ::= \_\_\_\_\_\_\_\_ \mid C \ e \ \mid v \ C
\]

A redex, shorthand for “reducible expression,” is an expression that can take a step according to a reduction rule. A value \((v)\) is either an integer or a closure:

\[
v \in \text{Values} ::= i \mid \langle \lambda x . e , \rho \rangle
\]
Closures are abstractions whose free variables are bound in the environment ($\rho$). An environment is a set of bindings which map variables to values, e.g. \{x $\mapsto$ v\}. Environments are defined by the following grammar:

$$\rho \in Environments ::= \emptyset \mid \{x \mapsto v\} \cup \rho$$

We use $\cup$ to denote the union of two disjoint sets. Notice that this requires all variables to have unique names.

An ECD machine state $(S)$ is represented by a triple containing a control—an expression, an environment, and a dump—a call stack. The evaluation rules for the ECD machine are defined in Figure 2.1. (VAR) handles variable lookup by extracting values out of the environment $\rho$. (LAM) creates a closure by duplicating $\rho$. (APP) applies a closure to a value. We reinstate the closure’s environment $\rho'$ extending it with a new binding \{x $\mapsto$ v\}. Notice that control is transferred to $e$ and the control and environment $(C, \rho)$ are saved on the call stack using $\cdot$ for concatenation. (RET) returns control to a previous context. The context $C$ and environment $\rho'$ are popped off the call stack and reinstated in their respective places. Notice $v$ is placed in the $\sigma$ of $C$.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
<th>$\rightarrow$</th>
<th>Next Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>(VAR)</td>
<td>$\langle C[x], \rho, \sigma \rangle$</td>
<td>$\rightarrow$</td>
<td>$\langle C[\rho(x)], \rho, \sigma \rangle$</td>
</tr>
<tr>
<td>(LAM)</td>
<td>$\langle C[\lambda x. e], \rho, \sigma \rangle$</td>
<td>$\rightarrow$</td>
<td>$\langle C[\lambda x. e, \rho'], \rho, \sigma \rangle$</td>
</tr>
<tr>
<td>(APP)</td>
<td>$\langle C[\lambda x. e, \rho'] v, \rho, \sigma \rangle$</td>
<td>$\rightarrow$</td>
<td>$\langle e, {x \mapsto v} \cup \rho', (C, \rho) \cdot \sigma \rangle$</td>
</tr>
<tr>
<td>(RET)</td>
<td>$\langle v, \rho, (C, \rho') \cdot \sigma \rangle$</td>
<td>$\rightarrow$</td>
<td>$\langle C[v], \rho', \sigma \rangle$</td>
</tr>
</tbody>
</table>

Figure 1: ECD machine

Evaluation of an expression is defined by a function:

$$\text{eval}_{ECD}(e) = (v, \rho) \text{ if } \langle e, \emptyset, [] \rangle \rightarrow^* \langle v, \rho, [] \rangle$$

The initial machine state contains an empty environment and call stack. If the machine converges after a series of finite steps, the result is $(v, \rho)$. As an example, a trace of $\text{eval}_{ECD}(\lambda x. \lambda y. x \ 1 \ 2) = (1, \emptyset)$ looks like:

$$\langle \lambda x. \lambda y. x \ 1 \ 2, \emptyset, [] \rangle \rightarrow \langle \lambda x. \lambda y. x, \emptyset, [] \rangle \rightarrow \langle \lambda y. x, \{x \mapsto 1\}, \emptyset, [] \rangle \rightarrow \langle \lambda y. x, \{x \mapsto 1\}, \emptyset, [] \rangle \rightarrow \langle x, \{x \mapsto 1, y \mapsto 2\}, \emptyset, [] \rangle \rightarrow \langle 1, \{x \mapsto 1, y \mapsto 2\}, \emptyset, [] \rangle$$

### 2.2 ECDH machine

In this section, we modify the ECD machine to expose the heap—a run-time representation of memory. As a result, the evaluation rules allocate values in the heap and dereference pointers as needed.

We define the heap ($\mu$) as a set of bindings that map heap addresses ($a$) to values. Environment variables are modified to map to heap addresses. We let $\text{Dom}(\mu)$ refer to the heap addresses in $\mu$, and $\text{Dom}(\rho)$ refers to the variables in $\rho$.

The semantics of the ECDH machine make allocation and dereferencing explicit. However, values are never deallocated. We can specify values that are no longer required for evaluation,

**Definition 2.1** (Garbage) If $\langle e, \{x \mapsto a\} \cup \rho, \sigma, \{a \mapsto v\} \cup \mu \rangle \rightarrow S'$, then the binding \{x $\mapsto$ a\} and its associated storage \{a $\mapsto$ v\} are garbage iff $\langle e, \rho, \sigma, \mu \rangle \rightarrow S'$. 

3
2.3 ECDH* machine

The missing functionality of the ECDH machine is deallocation of values—similar to popping a stack frame. The (Ret) rule is modified in the ECDH* machine to reflect this. However, we must be sure to deallocate the same value that the lambda allocated. To keep track of the address, whenever we allocate a value and transfer control to the body of a lambda, we push the address on the dump. When we return, we retrieve the address to deallocate from the dump.

![Figure 2: ECDH machine](image)

Evaluation is defined as,

$$\text{eval}_{ECDH}(e) = (v, \rho, \mu) \text{ if } \langle e, \emptyset, [], [] \rangle \rightarrow^{*} \langle v, \rho, [], \mu \rangle$$

**Definition 2.2** A state $S = \langle e, \rho, \sigma, \mu \rangle$ is stuck if there does not exist another state $S'$ such that $S \rightarrow^{*} S'$ and $e$ is not a value.

3 Typed stack allocation

The judgement

$$\Gamma \vdash t : \phi \tau$$

may be read as “Under the assumptions $\Gamma$, term $t$ observes the effects $\phi$ and yields a value of type $\tau$.” An effect can be thought of as reading a value from a variable. Thus, the effect $\phi$ of an expression $e$ may be thought of as the set of variables and their corresponding values that may be accessed upon evaluating $e$. A key insight to this type system is that we annotate the function type with a set of effects:

$$\tau \xrightarrow{\phi} \tau$$
This allows us to type functions with respect to the variables that they may access during evaluation. We define $\text{fv}(\tau)$ as a function that returns the set of free effects in the type $\tau$. $\emptyset$ denotes an empty set of effects.

The Const rule handles constants and we assume there is some function $\Delta$ that maps constants to base types. Under the Var rule, we observe the effect $x$ which yields the final effect type $x\tau$. The Lam rule handles lambdas. They key idea here is that the variable $x$ is not an observable effect within $\phi$. Notice that $x$ may be observable within $\phi$ though.

App handles applications where the observable effects are gathered in the final effect type $\phi'\psi'\tau'$. 

3.1 $\lambda_S$ Examples

The type and effect system of $\lambda_S$ ensures that we will never reach a stuck state in the ECDH* machine. This means that all well-typed $\lambda_S$ terms will allocate all values on the stack and evaluate to some value. Let us examine terms that may or may not classify as well-typed.

The identity function $\lambda x : \text{bool}.x$ is well-typed and it is safe to deduce that $x$ may be allocated on the stack because its only use occurs before the corresponding lambda returns. In fact, all terms of type $\tau \rightarrow \tau$ may safely allocate their parameters on the stack. In $\lambda x : \text{bool}.w$, $x$ is never used. Thus, it is safe to allocated on the stack.

Terms of the type $\tau_1 \rightarrow \tau_2 \rightarrow \cdots \tau_n$ are a bit more interesting. The term $\lambda g : \text{int}.\lambda y : \text{int}.x$ is well-typed, but $\lambda x : \text{int}.\lambda y : \text{int}.x$ is not. In $\lambda g : \text{int}.\lambda y : \text{int}.x$, both $g$ and $y$ are never used; it is safe to allocate them on the stack. However, in $\lambda x : \text{int}.\lambda y : \text{int}.x$, $x$ is used after the lambda that allocated $x$ returns. Understand how we fail to construct a derivation for this term:

\[
\frac{x : \text{int}.y : \text{int} \vdash x : \emptyset \text{int} \quad \text{VAR}}{x : \text{int} \vdash \lambda y : \text{int}.x : \emptyset \text{int} \quad \text{LAM}} \quad \frac{\vdash \lambda x : \text{int}.\lambda y : \text{int}.x : ? \quad \text{FAIL}}
\]

The reason why we cannot apply the last Lam rule is because $x \in \text{fv}(\text{int} \xrightarrow{\emptyset} \text{int})$. This violates the constraint in Lam that is crucial for ensuring safe stack allocation of lambda.
Function applications offer the most interesting derivations despite the simplicity of APP. It may be obvious to notice that the term \( \lambda x : \text{int}.(\lambda y : \text{int}. y x) \) is well-typed and will evaluate properly in the ECDH*/ machine. However, consider the faulty derivation of the term \( \lambda y : \text{int}.(\lambda x : \text{int} \rightarrow \text{int}. y \lambda g : \text{int}.x) \): The derivation fails when trying to apply \( \text{Lam} \) because of the same reason in the previous failed derivation. In this example, the argument value \( \lambda g : \text{int}.x \) applied to \( \lambda y : \text{int} \rightarrow \text{int}.y \) is said to have escaped the application. Let us examine a case where arguments in an application do not escape. To make things simple, we are going to assume that a binding for \( w : \text{int} \) already exists in our typing assumptions:

\[
\begin{align*}
\text{VAR} & \quad \Gamma \vdash w : [w] \text{int} \\
\text{LAM} & \quad \Gamma \vdash \lambda y : \text{int} \rightarrow \text{int}.w : \emptyset (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \\
& \quad \Gamma \vdash \lambda g : \text{int}.x : \emptyset \text{int} \rightarrow \text{int} \\
\text{APP} & \quad \Gamma \vdash \lambda x : \text{int}.(\lambda y : \text{int} \rightarrow \text{int}.w \lambda g : \text{int}.x) : \emptyset \text{int} \rightarrow \text{int}
\end{align*}
\]

Here we can see that effects from the non-escaping argument are not collected.

### 3.2 Soundness

Type soundness states that a program will never enter a stuck state during execution. We attempt to prove type soundness in the style of Wright and Felleisen [8] utilizing the standard Progress and Preservation lemmas.

The environment \( \rho \) and the heap \( \mu \) must be consistent. That is, if we typecheck with respect to assumptions about the types of values in the heap, we must ensure that we evaluate with a heap that conforms to these assumptions.

**Definition 3.1** A heap \( \mu \) is said to be well typed with respect to a typing context \( \Gamma \) and environment \( \rho \), written \( \Gamma \vdash \rho, \mu \), if for all \( x : \tau \in \Gamma \), \( x \in \text{Dom}(\rho) \) and \( \rho(x) \in \text{Dom}(\mu) \) and \( \mu(\rho(x)) : \tau \).

Intuitively, a heap \( \mu \) is well typed if every value in the heap has the same type predicted by the assumptions in the typing context \( \Gamma \).

The traditional Progress lemma states that a well-typed term does not get stuck; either it is a value, or it can take a step according to the evaluation rules. For our purposes, we will rephrase Progress to mean that a well-typed state does not get stuck; either it contains a value, or it can take a step in evaluation. This means that we must define what it means for a state to be well typed. Figure 5 contains the typing judgements that describe what it means for a state to be well typed. We also require that the heap be well typed to rule out stuck states as being possible well typed states.

**Lemma 3.2** (Progress) If \( \Gamma \vdash s : \tau \) where \( s = \langle e, \rho, \sigma, \mu \rangle \), then either \( s \rightarrow s' \) or \( e \) is a value and \( \sigma \) is empty.

**Proof.** The proof proceeds by induction on the derivation of \( e : \tau \).
• Const
  From the Const definition we have \( \cdot \vdash \langle c, \rho, \sigma, \mu \rangle : \tau \). So either \( \cdot \vdash \langle c, [], \sigma, \mu \rangle : \tau \) if \( c \) is a value, or we may take a step in the form of \( \langle c, \rho, (C', \rho', a) \cdot \sigma, a \cdot \mu \rangle \leftarrow \langle C[c], \rho', \sigma, \{a \mapsto \bullet \} \cdot \mu \rangle \) according to (Ret).

• Var
  Trivial since no variable is well-typed in an empty context.

• Lam

• App

Preservation states that if a well-typed term takes a step in evaluation, then the resulting term is also well-typed.

**Lemma 3.3** (Preservation) If \( \Gamma \vdash \langle e, \rho, \sigma, \mu \rangle : \tau \) and \( \langle e, \rho, \sigma, \mu \rangle \rightarrow \langle e', \rho', \sigma', \mu' \rangle \), then \( \Gamma \vdash \langle e', \rho', \sigma', \mu' \rangle : \tau \).

**Definition 3.4** (Well-typed Contexts) A context \( C \) is well-typed, \( C : \tau \rightarrow \tau' \), if and only if \( e : \tau \) and \( C[e] : \tau' \).

**Lemma 3.5** (Unique Decomposition) If \( e : \tau \) then either \( e \) is a value or there exists a context \( C \) and redex \( r \) such that \( e = C[r] \) and \( C : \tau \rightarrow \tau \) and \( r : \tau' \).

**Proof.** The proof proceeds by induction on the derivation of \( e : \tau \).

• Const, Lam – Trivial since \( e \) is a value.

• Var

• App
  – Suppose \( e_1 \) and \( e_2 \) are values. Then let \( C = \circ \) and \( r = e_1 e_2 \). So \( e_1 e_2 = C[r] \), \( C : \circ \rightarrow \circ \), and \( r : \circ \).
  – Suppose \( e_1 \) is a value and \( e_2 \) is not a value. From the induction hypothesis there is a \( C' \) and \( r \) such that \( e_2 = C'[r] \), \( C' : \tau \rightarrow \tau \), \( r : \tau' \), and \( r \) is a redex. Then \( C = e_1 C' \) so \( e = C[r] \) and \( C : \tau' \rightarrow \circ' \).

**Conjecture 3.6** (Type Soundness) If \( \Gamma \vdash s : \tau \) where \( s = \langle e, \rho, \sigma, \mu \rangle \), then either \( s \rightarrow s' \) or \( e \) is a value of type \( \tau \) and \( \rho, \sigma \) are empty.

4 Related Work

Georgeff’s [4] work has been noted to be the first in trying to determine stackability. The main contribution of his work is an extended SECD machine that allows partial application of abstractions. This is feasible by a closure of the form \( \langle \lambda x.e, \rho_1, \rho_n \rangle \) where \( \rho_1 \) is the local-environment and \( \rho_n \) is the non-local environment. Evaluation of nested lambda abstractions collect the bindings within the local-environment. An application of the form \( (e_1 e_2) \) proceeds if \( e_1 \) is closure where the body of the lambda abstraction is anything but a lambda abstraction. The local environment bindings are appended to the non-local environment as a whole upon function application. Stackability is retained for “simple expressions” that may have the type \( \tau \rightarrow \tau \rightarrow \cdots \tau \).

Banerjee and Schmidt [3] build on Georgeff’s work by noticing that stackability holds for expressions that may not classify as simple expressions. They observe that the essence
of stackability is that the shape of the environment must remain the same before and after execution of an expression. Specifically, stackability may be lost when the result of an application is a closure. Therefore, the environment trapped within the closure must be a subenvironment of the original environment. They then define the set of stackable expressions by examining dynamic semantic trees. This study yields a dynamic criterion that must hold for stackable expressions. Towards providing a static analysis to determine stackability, Banerjee and Schmidt then use a technique called closure analysis to determine what closures may be returned when evaluating an expression. They then modify the static analysis to ensure that dynamic criterion is held.

Goldberg and Park [5] use an abstract interpretation-based escape analysis to determine stackability. Arguments may be stack allocated if they do not escape their function call. It is not yet clear to me what set of expressions they identify may retain stackability.

Data-flow analysis refers to techniques that allow us to infer information about values such as points of allocation or access [1]. Unfortunately data-flow analysis depends on control-flow analysis which traces possible execution paths of a program. In functional programming languages, data-flow and control-flow analysis is difficult because they depend on each other[7][6].

5 Conclusion

We have defined an abstract machine that models stack allocation and specified a type and effect system for describing memory-safe programs with respect to the machine.
References


