



Feudal C

*Automatic Memory Management in a Static Type System
With Low Runtime Overhead*

Dan Bonachea, Carol Hurwitz, Scott McPeak
{bonachea, hurwitz, smcpeak}@cs.berkeley.edu

Abstract:

This paper defines a new imperative language called Feudal C which is essentially a restricted subset of C. Feudal C provides automatic memory management and statically prevents dangling references and memory leaks with very low runtime overhead. The key enabling insight is that memory management information in typical programs is *static*. This paper presents the syntax, typing rules and operational semantics for Feudal C, and proves memory safety for the language.



Feudal C

Automatic Memory Management in a Static Type System With Low Runtime Overhead

Abstract:

This paper defines a new imperative language called Feudal C which is essentially a restricted subset of C. Feudal C provides automatic memory management and statically prevents dangling references and memory leaks with very low runtime overhead. The key enabling insight is that memory management information in typical programs is *static*. This paper presents the syntax, typing rules and operational semantics for Feudal C, and proves memory safety for the language.

1 Introduction

In this section, we describe the goals of this work.

1.1 Automatic Memory Management

Memory management, in this context, means the method for deciding when to deallocate dynamically allocated memory.

1.1.1 Memory Management Errors

There are two kinds of memory management errors that can occur in unsafe programming languages (such as C):

Memory Leak	If the program deallocates an object too late (or not at all), this is called a memory leak. For example, memory that has been allocated but is no longer reachable, can never be used or deallocated and is therefore considered leaked. The effect of this error is that the program uses more memory than necessary. In long-running programs, such as operating systems, memory leaks are often the cause of eventual crash.
Dangling Reference	If the program deallocates an object too soon, such that a pointer still points to the deallocated space, this is called a dangling reference. The effect of this error is unpredictable; it might have no effect, or the dangling pointer might mysteriously change some data, eventually leading to a crash or other incorrect behavior.

Both kinds of memory management errors are very difficult to debug because they are untrapped errors. It is often very hard to localize the cause of such an error, even once it is detected. Still worse, it is often not easily detected -- that is, until the product is in the user's hands. Furthermore, memory safety is crucial to the integrity of any runtime data structures in the same address space, and therefore is often the first step in languages that provide other guarantees about dynamic behavior through runtime checking (such as languages with runtime array bounds checking). A language system that automatically prevents both types of errors is said to be memory safe.

1.1.2 The State of the Art

Currently, there are three basic approaches to memory management, each of which attempts to solve the above memory management errors:

Manual	The programmer explicitly inserts, into the program text, calls to a deallocation routine. This method has the least runtime cost, but is the most susceptible to memory management errors. Most C and C++ programs use this method.
Garbage Collection	The runtime system decides at runtime when an object is no longer being used, and deallocates it then. This method has a high runtime cost, including (for non-incremental collectors) occasional "pauses" in execution, which are very detrimental to user interactivity. It also places restrictions on the form of the runtime state, because the collector must be able to distinguish pointers from other data (this restriction is the major impediment to the use of garbage collection in C and C++ programs). A correctly implemented collector will prevent both kinds of memory management errors. Java, a few C and C++ systems, and most Lisp systems use this method.
Reference Counting	All objects automatically maintain a count of the number of pointers pointing to it; when the count

reaches zero, the object is deallocated. In spirit, this is similar to garbage collection, but with significantly different performance characteristics. Reference counting eliminates the execution pauses, but introduces additional memory usage requirements, because it must store a count with every object. Reference counting cannot cause dangling references, but it will leak cyclic structures (unless the programmer breaks cycles manually). A few Lisp systems use reference counting.

Each of these approaches trades performance for safety, and vice-versa.

1.1.3 An Attractive Alternative: The Type System

The type system is an attractive place to solve problems. It offers a convenient and well-understood mechanism for systematic static analysis, without impacting runtime performance.

However, for a static type system to work, the concepts codified as types must be *static* properties of the program (i.e., they do not change during execution). If they are not, the type system will only hinder programming. As an example, consider using the type system to distinguish between ints and floats, versus using it to distinguish between evens and odds. In the former case, it is critical that the runtime system know which is which, because it uses different hardware instructions to implement them. Further, it is very unusual for a program to want a variable to behave as an int some of the time, and a float at other times. In contrast, odds and evens need not be distinguished at runtime, and variables are very likely to need to assume both kinds of values during execution.

With the preceding in mind, we can formulate two key questions for evaluating a type system:

1. Are the concepts, represented by the types, *useful* at runtime?
2. Are the distinctions *static*?

It is clear that memory management information is useful at runtime; deallocation is a runtime activity. Further, *it is the central thesis of this work that, in fact, memory management information is static* (for most programs, in most cases). If this is true, the type system is clearly an ideal place to do much memory management activity. In subsequent sections of this paper, we will illustrate and develop this idea, and exhibit such a type system.

1.2 Design Evident in Code

A further benefit to using the type system, especially in a language with explicit type declaration (as opposed to type inference), is that it makes features of the design evident in the code. Using again the example of distinguished types for ints and floats, most of the time the choice of which to use reflects a choice *already made* during the design process (whether that process was explicit or implicit). Someone reading the code at a later time can learn pieces of the design, if not the whole design, by observing the types used.

Memory management design is arguably even more crucial than low-level details like number representation. A type system which created and enforced memory management concept distinctions would necessarily carry with it information about this aspect of the program's design. Just writing the type annotations in the first place would encourage thinking clearly about memory management design. In contrast, many popular (albeit ad-hoc) design methodologies postpone such decisions, to the detriment of overall code quality -- a deallocation call hastily inserted is probably wrong.

Therefore, the design of the memory management type system was driven in part by the desire to model what we believe to be good design principles, and to make the code faithfully reflect that design.

1.3 Static Memory Management Information

The usability of the Feudal C language depends crucially on the assumption that memory management information is static. Lacking an extensive study of existing programs, we can sketch some reasons one should believe this is true. First, consider that many programs allocate data on the stack when possible as a performance optimization; this is clearly static memory management. Further, typical C++ style has deallocation calls in object destructors; again, a static property. Even more general C style rarely has calls to **free** inside an **if** – the decision to deallocate has been made statically by the programmer, because the program design supports it.

The main point here is that we observe that typical programs (using explicit memory management) do not have complex decision procedures to decide when to deallocate memory. As such, it is sensible to try to put the memory management reasoning, normally done by the programmer, into a static vehicle like the type system.

1.4 Related Work

LCLint [LC99] is a static analysis tool that looks for common programming errors and reports them to the user. It also has a system of annotations for tagging (among other things) pointers with keywords that identify how they are used. LCLint's "owner" tag means

roughly the same thing as ours, and its “dependent” tag corresponds loosely to our “serf.” However, LCLint makes no attempt to check that no dependent pointers are pointing to an object when it is deallocated (allowing dangling references to go undetected).

Our type system is the first we know of that statically prevents both kinds of memory management errors in a language with explicit dynamic allocation/deallocation.

2 Fundamental Concepts

2.1 Owners

The most fundamental concept of our work is the notion of ownership. Simply put, ownership is the obligation to deallocate. For example, if object A creates object B in its constructor, retains a pointer to B throughout its lifetime, and deallocates B in its destructor, we say that A owns B. Ownership is transferable; if object A creates B, it owns B. Later, if A gives a copy of its pointer to B to another object, C, and then nullifies its own pointer to B, we say that A has transferred ownership of B to C.

We can generalize ownership to include function invocations; we say that a function invocation owns B if it created B, and therefore it should either transfer ownership, or deallocate B when it returns. We also consider a function invocation to own function invocations it has directly called. If foo() calls bar(), then an invocation of foo() will own an invocation of bar(). This kind of ownership is not transferable.

In all cases, there is one property of ownership that must be maintained:

- * All allocated objects must have *exactly* one owner.

This immediately implies that an owner must either deallocate the ownee, or transfer ownership to another, before it is itself deallocated. The maintenance of this property is the primary mechanism for preventing memory leaks.

2.2 Serfs

All pointers that are not owners are called “serfs”¹, suggestive of working on but not owning something. There is one property all serfs must have:

- * There must be no serfs pointing at an object when it is deallocated.

The maintenance of this property prevents dangling references.

2.3 The Tree

A key insight in our work is the notion of the “owner tree”. The owner tree is the tree obtained by considering all ownership relationships, including the current call stack (active function invocations). There remains some debate as how best to formulate the root of the tree; one possibility is to call the global variables the root, and main() the principal child of the globals. Figure 2.3 is an example of such a tree.

In Figure 2.3, there are several familiar structures. For example, bigArray is a local variable of main(), and itself owns three elements. dl1, dl2, and dl3 are a doubly-linked list. The dashed lines are serf pointers; bar() has a pointer to the second element of the array, and the second invocation of foo() has a pointer to the fooLocalObj allocated by the first invocation, as well as the one it allocated itself.

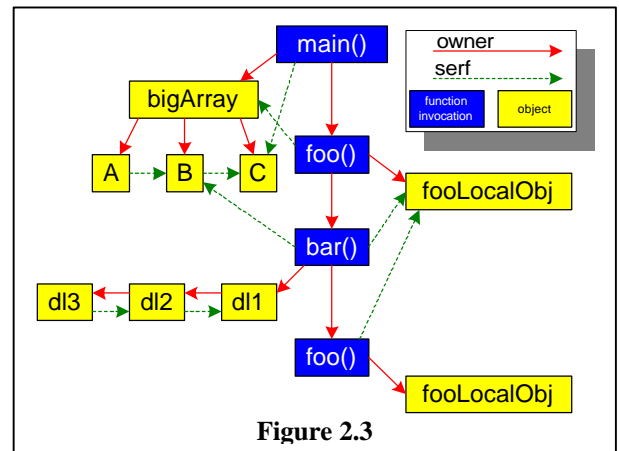


Figure 2.3

2.4 The Restricted Type System

Appendix D outlines the full range of owner and serf pointer varieties that we envision for a real implementation of this language.

Due to time limitations for the project, we chose to focus on the most restricted variety of pointers which have no runtime component; the “restricted” owner and serf pointers. In future work, we plan to expand our inquiry to consider at least the range of types presented above.

¹ Serf is synonymous with “peasant”: those who, in feudal times, worked the land in exchange for the protection offered by the land’s owner. Note that, in our model, nothing prevents owners from doing work also.

3 Feudal C Syntax

This section provides an overview of the syntax of Feudal C. The complete syntax appears in Appendix A.

3.1 General Language Information and Syntax

The language was designed to be as simple as possible to streamline the proofs, while still retaining the inherent complexity of the issues to be studied. Many of the language features of C were left out without loss of generality – for example, block scopes, global variables, expressions with side effects, and (non recursive) typedef can all be simulated in our language through straightforward program transformations (although some of these transformations may expand the code exponentially). Other restrictions that were added impose more fundamental limitations on the expressive power of the language, such as the lack of arrays, structures, function pointers, the address-of operator and recursive types. However, the hope is that these features may be reintroduced to the language once the essential memory management issues have been studied in relative isolation.

3.2 Rules of Thumb

There are several general "rules of thumb" enforced by the syntax and typing rules which are helpful to remember when studying Feudal C:

- Serf pointers may not be stored in the heap or returned from functions
- Owner pointers may not be passed as parameters to functions
- Serf pointers cannot be dereferenced to obtain owner pointers

Together, these rules enforce the owner-tree property of the program that makes it possible to perform automatic memory management statically. Specifically, they ensure that owner pointers always point "down" and serf pointers never point "down". In section 6, we will see in more detail why these rules of thumb are important to the proof of memory safety for the language.

3.3 Syntax

Implicitly pre-defined sets:

ω - The set of all non-negative integer literals, i.e. $\{0, 1, \dots\}$

Loc – The set of all legal variable names

Fname – The set of all legal function names (defined s.t. $\text{Loc} \cap \text{Fname} = \emptyset$)

Aexpr ::= n <Lvalexpr> <Aexpr> op <Aexpr>	$n \in \omega, \text{op} \in \{+, *, \&\&, , ==, !=\}$
Lvalexpr ::= x * <Lvalexpr>	$x \in \text{Loc}$
TypeAux ::= int <TypeAux> owner *	
Type ::= <TypeAux> <TypeAux> serf *	
DeclSeq ::= <Type> x ; <DeclSeq> ϵ	$x \in \text{Loc}$
GdeclSeq ::= <GdeclSeq> ; <GdeclSeq> <Type> f(<DeclSeq>) <DeclSeq> <Com> ; <RetCom>	$f \in \text{Fname}$
RetCom ::= return x	$x \in \text{Loc}$
Com ::= <Com> ; <Com>	
if <Aexpr> then <Com> else <Com> while <Aexpr> do <Com> skip	
<Lvalexpr> = <Aexpr> <Lvalexpr> := <Lvalexpr>	
x = f(<Aexpr>, ..., <Aexpr>) x := f(<Aexpr>, ..., <Aexpr>)	$x \in \text{Loc}, f \in \text{Fname}$
x := new (<Type>)	$x \in \text{Loc}$
delete (x)	$x \in \text{Loc}$
Program ::= <GdeclSeq>	

Figure 3.3

Figure 3.3 gives an abbreviated syntax for Feudal C. The complete syntax appears in Appendix A. The only data types are integers and chains of pointers to integers, and the syntax is similar to C. Pointers may be either of the owner or serf variety, but notice the syntax restricts chains of pointers to contain at most one serf pointer, which must appear last. This enforces the restriction that serf pointers may only exist in the stack – it's not possible to declare a type that includes a serf pointer on the heap.

The syntax for variable declarations and function definitions is identical to C, with the only restriction being that the last command in every function must be a **return** statement, and this is the only place that **return** can occur (this was done without loss of generality to simplify the proof).

3.3.1 Commands

The language offers commands similar to those found in most other imperative languages: conditional (**if-then-else**), looping (**while-do**), function call and **return**, assignment, and dynamic memory (de)allocation (**new/delete**). Notice that unlike C, function call, assignment, and (de)allocation may not appear within expressions. Furthermore, with the exception of assignment, all commands that modify an L-value require a local variable rather than a general L-value expression (again, this was done for simplicity and without loss of generality).

Feudal C contains one very special and unique command that characterizes its memory management strategy: the transfer assign, denoted by the operator (`:=`). As we shall later see, this command executes the transfer of memory ownership from one owner pointer to another, nullifying the source pointer in the process. These semantics are pivotal to the single ownership property of the language, which is embodied by the single owner lemma proved in section 6. The syntax for the operator was intentionally chosen as different from the standard C "copy" assignment (which also appears in the language), to emphasize the nullification of the right operand.

The (`:=`) operator also appears in the new and function call transfer commands, since these also involve assigning ownership of memory to an owner pointer, although the concept of ownership release is less important in these cases.

3.3.2 Expressions

Expressions in Feudal C consist of integer literals, variables on the stack and dereferenced pointers, combined using typical binary arithmetic and logical operators. The typing rules forbid pointer arithmetic, and expressions never have side effects. We don't concern ourselves with the specific semantics of the operators, however we do follow the C convention of treating integer expressions as Boolean values (where zero indicates false and non-zero indicates true) in the appropriate places (the guard expression in the **while** and **if** constructs).

3.3.3 Sample Code

```

1. int owner* owner* allocSmallList(int val) {
2.     int owner* x;           // variable declarations
3.     int owner* owner* y;
4.     int serf* z;

5.     x := new int;           // allocate an integer on the heap
6.     *x = val;               // store val there
7.     z = x;                  // get the address in a serf pointer
8.     *z = *x + 5;            // change the stored value again
9.     y := new int owner*;    // create a pointer on the heap
10.    *y := x;                 // this nullifies x
11.    return y;                // safely returns allocated space!
12. }

```

Figure 3.3.3A

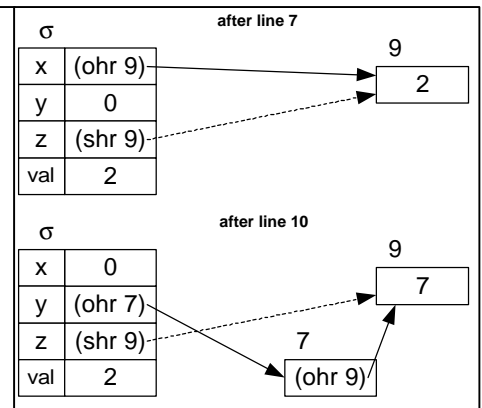


Figure 3.3.3B

Figure 3.3.3A presents a simple program fragment written in Feudal C which allocates a small list on the heap and returns it. Figure 3.3.3B graphically demonstrates the behavior of this code. Notice on line 7 how we use the copy assignment command to create a serf pointer alias to the memory cell we allocated, and on line 8 we use the serf pointer to change the value stored in that cell. Also note on line 10 how we use the transfer assignment command to transfer ownership of the memory cell to *y nullifying x. (Nullifying x means we set it equal to zero – the pointer value of zero is not a valid address and represents an uninitialized pointer. Attempting to dereference null is assumed to raise a trapped runtime error, and is not dealt with further in our language). Note that in line 11 we safely pass ownership of the space we've allocated out of the function and transfer it to the function's caller. This is one of the key features in Feudal C that addresses a severe limitation in C. Many functions need to pass information back to their caller in a memory buffer, but in languages without automatic memory management (such as C) it's inadvisable to pass dynamically allocated space out of a function (because it can easily become a memory leak). The static memory management features of Feudal C successfully address this problem.

4 Feudal C Typing Rules

This section provides an overview of the typing rules of Feudal C. The complete typing rules appear in Appendix B.

4.1 Feudal C Typing Rules

Technically speaking, Feudal C is a safe language, with an explicitly typed, first-order type system. The complete typing rules for Feudal C are presented in Appendix A. We will now examine a few of the more interesting rules. The typing rules are expressed in a vertical form that is typical of language research, where premise judgements appear above the horizontal line and the conclusion judgement appears below the line. See [Ca96] for details.

4.1.1 Judgements

$\vdash F$	F is a well-formed function environment (program)
$\vdash^F f$	f is a well typed function, given function environment F
$\Gamma \vdash^F c$	Command c is well typed, given type environment Γ and function environment F
$\Gamma \vdash rc : \tau$	Return command rc returns type τ , given type environment Γ
$\Gamma \vdash e : \tau$	Expression e has type τ in type environment Γ

Figure 4.1.1

Figure 4.1.1 illustrates the judgements that appear in the typing rules and their meaning. The highest level object, the function environment F, represents the entire program as a set of function definitions corresponding to those given in the source code text. This function environment is used in the judgements for commands and functions, because their type derivations may include calls to other functions, which must be typechecked (expressions and return commands may not call functions).

4.1.2 Commands and Expressions

$\frac{\Gamma \vdash e : \text{int} \quad \Gamma \vdash^F c_1 \quad \Gamma \vdash^F c_2}{\Gamma \vdash^F \text{if } e \text{ then } c_1 \text{ else } c_2}$	$\frac{\Gamma \vdash x : \tau \text{ serf } * \quad \Gamma \vdash e : \tau \text{ owner } *}{\Gamma \vdash^F x = e}$
$\frac{\Gamma \vdash v : \tau \text{ owner } * \text{ serf } *}{\Gamma \vdash *v : \tau \text{ serf } *}$	$\frac{\Gamma \vdash v_1 : \tau \text{ owner } * \quad \Gamma \vdash v_2 : \tau \text{ owner } *}{\Gamma \vdash^F v_1 := v_2}$

Figure 4.1.2

Figure 4.1.2 illustrates a few of the typing rules for the language. The rule for the conditional is simple, yet clearly illustrates the general approach. The best way to read this rule is: “The command (**if** e **then** c_1 **else** c_2) is well typed in the type environment Γ and function environment F if and only if there exist well-formed typing derivations that show the expression e has type int in type environment Γ , and that commands c_1 and c_2 are well typed in type environment Γ and function environment F.”

The typing rule shown for serf pointer dereference (one of several given in the complete rules) illustrates the typing enforcement of the rule of thumb presented earlier - that serf pointers cannot be dereferenced to obtain owner pointers. The typing rule enforces this because it dictates that if v has type $\tau \text{ owner } * \text{ serf } *$, then the expression $*v$ will have type $\tau \text{ serf } *$, not type $\tau \text{ owner } *$.

The rule for the copy assignment command shows how owner pointers may be copied to serf pointers, as we did in line 7 of figure 3.3.3A. This is the only rule that allows the copying of data from one type to a different type. Specifically, it's important to note that there is no rule permitting the opposite command, that is, $x = e$ where x has type $\tau \text{ owner } *$ and e has type $\tau \text{ serf } *$. The typechecker will reject any programs for which a valid typing derivation cannot be constructed from the typing rules, so any programs containing such a command would fail to type check. This enforces the rule stated earlier that only the transfer assignment command(s) can be used to assign values into owner pointers. The typing rule for transfer assignment is also shown for comparison.

4.1.3 Function Call and Return

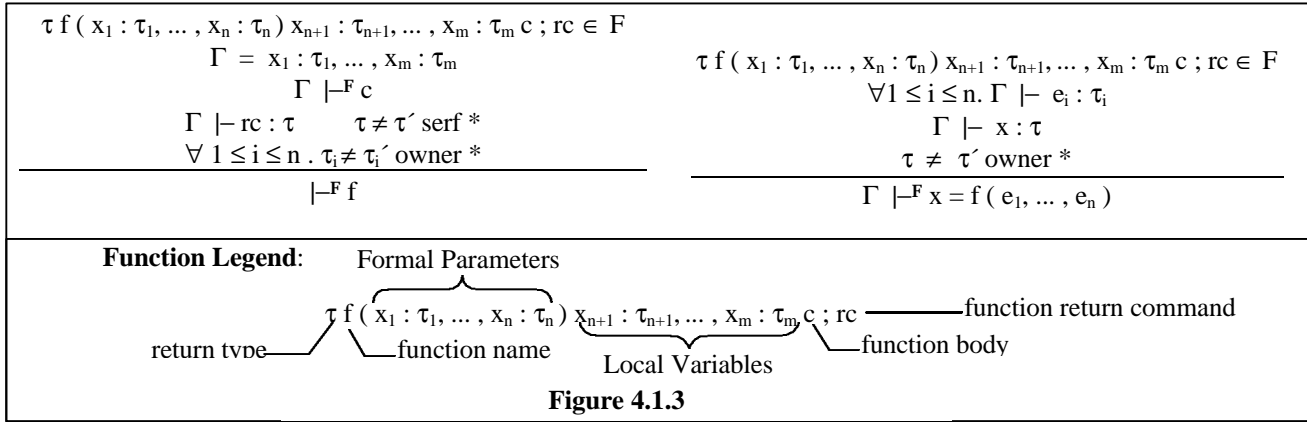


Figure 4.1.3 shows the typing rules that handle function definition and function call. Let's start by examining function definition. The highest-level typing rule for the judgement \vdash^F (the entire program is well typed) contains only the simple premise: $\forall f \in F . \vdash^F f$, so the function definition rule is the real powerhouse for typechecking the overall program structure. The rule states that in order for a function definition to be well typed, it must appear in F with the given τ (return type), $x_1 : \tau_1, \dots, x_n : \tau_n$ (formal parameter names and types), $x_{n+1} : \tau_{n+1}, \dots, x_m : \tau_m$ (local variable names and types), c (function body – a sequence of zero or more commands) and rc (a return command). The rule establishes a local type environment Γ , then uses it to check that the body of the function is well typed in Γ and F , and that the return command returns the proper type in environment Γ . This typing rule also enforces some of the rules of thumb discussed earlier which state that functions may not take owner pointers as parameters or return serf pointers.

The rule for the function call command is similar. In order for a function call to be well typed, the named function f must appear in F . The actual parameters (e_1, \dots, e_n) given in the function call must have types in the current type environment which match the types declared for the formal parameters of f . The variable which receives the return value of the function must have a type which matches the return type of the function. Finally, this rule only applies to functions which do not return owner pointers, because those that do must be called by the function call transfer command: $x := f(\dots)$ (whose rule is very similar).

5 Feudal C Operational Semantics

This section provides an overview of the operational semantics of Feudal C. Complete operational semantics appear in Appendix C.

5.1 Notation

Now let's introduce some new notation.

5.1.1 Semantic Entities

Definitions	Examples
Values $\equiv \omega \cup (\text{ohr } \omega) \cup (\text{shr } \omega)$	0 or 4 or (ohr 37) or (shr 5)
VarValueMap $\equiv P(\text{Loc} \rightarrow \text{Values})$	VVM = { (x, 5), (y, (shr 17)), (z, (ohr 17)) }
Frame $\equiv \text{VarValueMap} \times \text{Command} \times \text{Loc}$	$\sigma_n = \text{Frame}(\text{VVM}, c, x)$
Heap $\equiv P((\text{hr } \omega) \rightarrow \text{Values})$	H = { ((hr 16), 2), ((hr 30), (ohr 17)) }
States $\equiv \text{Frame list} \times \text{Heap} \times \text{Command}$	S = $\langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle$
Command $\equiv \text{Com} ; \text{RetCom}$	

Table 5.1.1

We will now examine the operational semantics for Feudal C. Table 5.1.1 gives definitions for the various semantic entities used to model the operational state of the abstract machine that executes programs in the language. The set Values represents the runtime values which may be generated by expression evaluation and stored or retrieved from memory locations and local variables. This set consists of the set of non-negative integers (ω), owner heap references (parameterized by ω), and serf heap references (also parameterized by ω). Heap references may be used to index into the heap, which is modeled as a partial function from heap references (parameterized by ω) to Values (the value stored in the named memory location). The choice of ω as the set used for parameterizing heap references is arbitrary, but it's important to note that for all $n \in \omega$, (ohr n), (shr n) and (hr n) all refer to the same memory location. The heap reference values were divided into serf and owner references only to clarify the proof – an implementation would not maintain this distinction (nor the distinction between integers and heap references, in all likelihood).

5.1.2 Program Execution

Program execution is modeled as a sequence of operational states (States) and the operational semantics define small-step mappings from a current state to the next state for each command. The state has three components: a list of stack frames, a heap, and the sequence of commands remaining to be executed. Each stack frame contains three elements, the most important of which is the VarValueMap, which is a partial function that maps names of local variables to their current values in the stack frame. The other two elements of the stack frame are a Command, which contains the remainder of the calling function's body (which plays the role of a return address for these semantics), and the name of the local variable in the previous stack frame which receives the return value for this function. We will examine both in more detail shortly.

5.1.3 Notation

$\text{eval}(e, \sigma, H) \Downarrow v$ <p style="text-align: center;">where $e \in \text{Aexpr}, \sigma \in \text{States}, H \in \text{Heap}$ and $v \in \text{Values}$</p>	<i>Expression evaluation (large-step)</i>
$\langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle \rightarrow \langle \sigma_1', \dots, \sigma_m' \mid H' \mid c' \rangle$ <p style="text-align: center;">where $\sigma_1, \dots, \sigma_n, \sigma_1', \dots, \sigma_m' \in \text{Frame}, H, H' \in \text{Heap},$ and $c, c' \in \text{Command}$</p>	<i>Command evaluation (small-step)</i>
$\sigma.\text{vvm} \equiv \text{first}(\sigma) \quad \text{where } \sigma \in \text{Frame}$	<i>VarValueMap extractor</i>
$\sigma.\text{cmd} \equiv \text{second}(\sigma) \quad \text{where } \sigma \in \text{Frame}$	<i>Return command sequence extractor</i>
$\sigma.\text{var} \equiv \text{third}(\sigma) \quad \text{where } \sigma \in \text{Frame}$	<i>Return value target variable extractor</i>
$\text{VVM}[v/x](y) \equiv \begin{cases} \text{VVM}(y) & \text{if } y \neq x \\ v & \text{if } y = x \end{cases} \quad \text{where } \text{VVM} \in \text{VarValueMap}$	<i>Variable value replacement</i>
$\sigma(x) \equiv \sigma.\text{vvm}(x)$	<i>Local variable value lookup</i>
$\sigma[v/x] \equiv \text{Frame}(\sigma.\text{vvm}[v/x], \sigma.\text{cmd}, \sigma.\text{var})$	<i>Local variable value substitution</i>
$H[v/x](y) \equiv \begin{cases} H(y) & \text{if } y \neq x \\ v & \text{if } y = x \end{cases} \quad \text{where } H \in \text{Heap}$	<i>Heap value replacement</i>

Figure 5.1.3

Figure 5.1.3 lists some additional notation and shorthand used in the operational semantics. The $\text{eval}(e, \sigma, H) \Downarrow v$ notation is used to denote a large-step evaluation of the expression e given stack frame σ and heap H (the evaluation does not return a modified heap or stack because expressions can't have side effects). The rules for the behavior of this construct appear as part of the operational semantics, and are similar to what one would expect – the only exception being the rule for dereferencing self pointers, which changes owner heap reference values to self heap reference values, as required by the typing rules.

The $\langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle \rightarrow \langle \sigma_1', \dots, \sigma_m' \mid H' \mid c' \rangle$ notation indicates a small-step command evaluation, where an initial state is passed through "one turn of the crank" to reach the correct next state – in most cases, the command just executed is removed from the input command c , and a new next command c' is specified in the next state. With the exception of memory allocation (which selects an arbitrary free address), the state change is completely deterministic.

The rest of Figure 5.1.3 explains the shorthand used in the rules to extract particular fields or values from the operational entities and to perform substitutions on the entities. These are a straightforward extension of standard programming language notation (see [Wi96] for details).

5.1.4 Helper Functions

$\text{TC} : \text{Heap} \times (\text{hr } \omega) \rightarrow P((\text{hr } \omega))$	<i>"Transitive Closure" - Returns all the heap addresses reachable from this heap address</i>
$\text{TC}(H, (\text{hr } n)) \equiv \begin{cases} \{(\text{hr } n)\} \cup \text{TC}(H, (\text{hr } x)) & \text{if } H((\text{hr } n)) = (\text{ohr } x) \\ \{(\text{hr } n)\} & \text{otherwise} \end{cases}$	
$\text{del} : \text{Heap} \times (\text{ohr } \omega) \rightarrow \text{Heap}$	<i>Deletes a chain of owner pointers in heap H, starting at address x, and returns modified heap</i>
$\text{del}(H, (\text{ohr } \text{addr})) \equiv H - \text{TC}(H, (\text{hr } \text{addr}))$	

Figure 5.1.4

Figure 5.1.4 defines two important helper functions used by the operational semantics. The transitive closure (TC) function takes a heap reference and returns the set of all heap references reachable from that heap reference. Notice the function constructs this set by following all the owner heap references it encounters (the only type of pointer value which may exist in the heap), stopping when it reaches an integer or a null pointer. $\text{del}()$ takes an owner pointer and deallocates all the memory owned by that pointer and its children

by subtracting the TC of that pointer from the heap. `del()` does the job of deallocating an entire owner tree in one step, given the root pointer.

5.2 Operational Semantics

Appendix C gives the complete operational semantics for the language; however, we will examine a few of the more interesting rules here to illustrate the concepts. The first rule given in Figure 5.2 is the operational rule for the dynamic memory allocation command. Notice how the first premise ensures that prior to executing the command the target variable doesn't point to anything (by looking up the value of x in the current stack frame and making sure it's zero). The second premise selects a heap reference value that's unused in the current heap H , and calls the parameter `newaddr`. The command does its work by modifying the σ_n and H elements for the next state – namely, by substituting the value `(ohr newaddr)` into the `VarValueMap` for the current stack frame (so future references to $\sigma_n(x)$ will return `(ohr newaddr)`), and by adding the new memory cell `((hr newaddr), 0)` to the heap H . Notice that this rule does not apply if x currently contains a valid owner pointer (if $\sigma_n(x) \neq 0$), and in fact under these circumstances, execution would halt because no rule would apply in this case. This is somewhat counter-intuitive – one might expect an automatic memory management system to simply deallocate the memory currently pointed to by $\sigma_n(x)$, and then perform the assignment. However, a trivial code transformation of inserting a `(delete x;)` command before every assignment to variable x accomplishes this. In Section 5.3, we discuss a simple preprocessor for the language that does exactly this. Several of the operational rules take the approach of checking certain conditions are true before executing, relying on the transformations of the preprocessor to implicitly ensure them if the programmer didn't explicitly state these operations. A real implementation would most likely rely on the preprocessor to either verify these conditions about the input program through static analysis or ensure them by inserting the necessary implicit commands in a conservative manner – thereby eliminating the need to perform any checks at runtime.

$\frac{\sigma_n(x) = 0 \quad (hr \text{ newaddr}) \notin \text{dom}(H)}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := \text{new}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n[(ohr \text{ newaddr})/x] \mid H \cup \{((hr \text{ newaddr}), 0)\} \mid c \rangle}$
$\frac{\begin{array}{l} f(x_1, \dots, x_m) x_{m+1}, \dots, x_p \text{ cbody} \in F_{op} \\ \forall 1 \leq i \leq m. \text{eval}(e_i, \sigma_n, H) \Downarrow v_i \quad \forall m+1 \leq i \leq p. v_i = 0 \end{array}}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x = f(e_1, \dots, e_m); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n, \text{Frame}(\{(x_1, v_1) \dots, (x_p, v_p)\}, c, x) \mid H \mid \text{cbody} \rangle}$
$\frac{\forall y \in \text{dom}(\sigma_n). y \neq x \Rightarrow \sigma_n(y) \neq (ohr z)}{\langle \sigma_1, \dots, \sigma_{n-1}, \sigma_n \mid H \mid \text{return } x \rangle \rightarrow \langle \sigma_1, \dots, \sigma_{n-1}[\sigma_n(x)/\sigma_n.var] \mid H \mid \sigma_n.cmd \rangle}$

Figure 5.2

The second rule of Figure 5.2 shows the operational semantics for the function call command. The primary behavior of this command is to construct a new stack frame, set the next command to be the called function body, and store the information necessary for function return. Notice the similarities in the first premise to the typing rules for the function definition and function call discussed earlier. In this case, we are selecting a function prototype from the operational function environment F_{op} , which is a direct translation of the function environment F constructed by the typing rules, with the type annotations stripped (they are no longer necessary) and the `Com` (body command) and `RetCom` (return command) elements merged into a single operational Command sequence `cbody` (we no longer need to distinguish these). The second rule premise evaluates each of the m actual argument expressions to values v_1, \dots, v_m , which become assigned to the formal parameters x_1, \dots, x_m in the `VarValueMap` for the stack frame we are constructing. The third premise defines values v_{m+1}, \dots, v_p to be zero and stores these values in the `VarValueMap` entries corresponding to the local variables x_{m+1}, \dots, x_p in the new stack frame, which initializes all local variables to zero. Next, we store the name of the local variable of the current stack frame that will receive the return value of the called function (x) into the new stack frame. Finally, we store the command continuation for the current function into the new stack frame and set the next command to be executed to be the body of the called function.

The third rule of Figure 5.2 illustrates the operational semantics for function return – its primary task is to pass the return value back to the calling function, pop the current stack frame, and jump back to the return address (command continuation). Recall that functions are permitted to return owner heap references, but the remaining owner heap references that are stored in the local variables of the returning function must have their owned memory freed before the function returns. Failure to do so would result in a memory leak, because there are no pointers in the previous stack frames or elsewhere in the heap which can reach this memory (as stated in the `NoSerfsDown` and `NoSerfsInTheHeap` properties presented in section 6), so this memory would become unreachable. The first premise checks that none of the local variables (except for possibly the one being returned) contain an owner heap reference. The next state for the return command is constructed by removing the top stack frame (σ_n) from the current state, substituting the return value $\sigma_n(x)$ into the correct variable in the previous stack frame ($\sigma_n.var$) and setting the next state command to be the command continuation that was stored in $\sigma_n.cmd$ when the function was called.

5.3 Preprocessor

The operational semantics require that certain things be true before some commands are executed. For example, the **return** command requires that all local owner pointers (except possibly the one being returned) are null before it executes. The syntax and type rules, however, do not enforce this. If typechecked programs are immediately interpreted by the operational semantics, legal programs could cause runtime errors (execution would halt because no operational rule would match the state).

To prevent memory management runtime errors, we assume the presence of a preprocessor that will perform some transformations on the input. These transformations are *not* depended upon for proving memory safety; they are, however, useful from the usability point of view. Toward this end, it is the intent that preprocessed programs will not cause runtime errors such as failing the above requirement regarding **return** commands. That is, the only runtime errors that should be possible after preprocessing are null dereference, out of memory, infinite loop, and arithmetic errors.

These are the transformations performed by the preprocessor.

Deallocate On Return	The preprocessor will insert a delete command before the return command for all local variables that are owner pointers, except the pointer being returned (if any).
Nullify Targets	The target of a transfer-assign command must be null (where transfer-assign here includes allocation and transfer function call). The preprocessor will insert a delete command to enforce this, just before the actual transfer assignment.
Alias Analysis	The delete command requires that there are no serf heap references pointing to the deallocated object, or any of its descendants. The preprocessor must determine which serfs in the current stack frame may point to the deallocated object or one of its descendants, and nullify those serf before the delete command. Note that any such serfs can only exist in the current stack frame, as proven by the NoSerfsDown invariant in section 6. This can be done by sophisticated (intra-procedural) static analysis, or by a conservative and simpleminded kill-all-serfs approach.

Some of these inserted commands are necessary (e.g. deallocation), and therefore should not count as runtime overhead. Others, such as nullification of aliased serfs, can be removed if the optimizer can determine that the serf isn't used after it is nullified (as, in fact, it should not be used). Thus, it is our claim that the introduction of the preprocessor does not significantly increase runtime costs.

6 Proof of Memory Safety

In this section, we provide the proof of memory safety for Feudal C. The proof refers repeatedly to the operational semantics, which appear in Appendix C in their entirety.

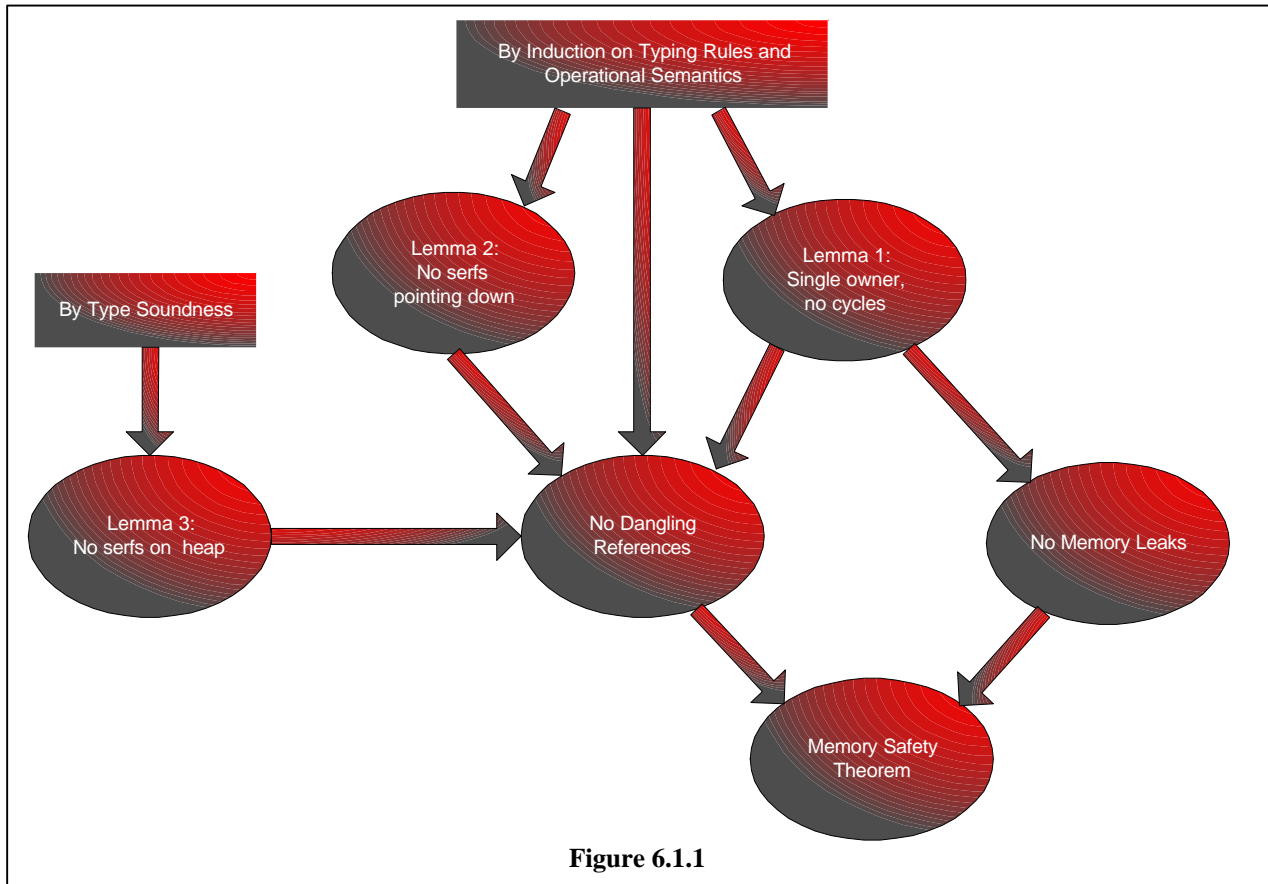
6.1 Proof Introduction

6.1.1 Overview

To prove the memory safety of the Feudal C language, we prove that it does not leak memory, and that dangling references do not occur. Note that throughout this discussion, we are assuming type soundness.

These theorems require three lemmas. First, we prove an invariant that every allocated cell has exactly one owner, and that there are no ownership cycles; this is proven by induction on the execution of the operational semantics. Then, we prove that there are no serfs pointing “down”, in the sense that they don't point to cells in the transitive closure of variables in lower stack frames; this is also proven by induction on the execution of the operational semantics. Finally, we show that type soundness implies that there are no serfs on the heap.

The first lemma then directly implies the invariant that there are no memory leaks. Then, with all three lemmas, and one final induction, we prove the invariant that there are no dangling references. Figure 6.1.1 provides a graphical depiction of this proof strategy.



6.1.2 General Approach

To prove invariants about the execution of Feudal C programs, we induct on the execution of the operational semantics. To fit this into the framework of well-founded induction, we need a partial order with no infinite descending chains. We can define such an order by augmenting states to include the ordered list of all previous states in a legal program execution, and letting the is-a-prefix relation be the predecessor relation. Then, we are doing well-founded induction on the state-list partial order, where the rules of the operational semantics define the means for extending one list of states to its successor, and the singleton list containing only the initial state is the base case. We will call this approach induction on the execution of the operational semantics.

We now describe the form this induction takes in each of the proofs on the operational semantics to prove that a given invariant holds for the entire execution of any legal program (where “legal” is defined to be a syntactically correct program that passed the typechecker). The first step is to prove the invariant of interest (P) holds for the base case of the initial execution state, which for any program in Feudal C is given by the following:

$$\text{InitialState} = \langle \text{Frame}(\{ (x, 0) \}, \text{return } y, y) \mid \emptyset \mid x = \text{main}(); \text{return } x \rangle$$

We assume for simplicity that all Feudal C programs define the function: `int main()`. Notice that if `main()` ever returns, execution will halt in the following final state: (the clauses involving `y` are just a dummy command and variable which are never used)

$$\text{FinalState} = \langle \text{Frame}(\{ (x, 0) \}, \text{return } y, y) \mid H \mid \text{return } x \rangle$$

Once we have proven $P(\text{InitialState})$, the second, inductive step is to prove that each rule in the operational semantics (and therefore every possible state change) preserves the invariant P . That is:

$$\forall S \in \text{State} . P(S) \wedge (S \rightarrow S') \Rightarrow P(S')$$

If we can prove both of these cases for the invariant P , then by induction on the execution of the operational semantics, the invariant holds for all states throughout the execution of the program, that is:

$$\forall F \in \text{GdeclSeq} . \vdash F \wedge P(\text{InitialState}) \wedge (\text{InitialState} \rightarrow^* \text{AnyState})_{\text{Fop}} \Rightarrow P(\text{AnyState})$$

6.2 Theorem Statement

First, we formally state the high-level invariants to be proven regarding memory safety:

6.2.1 Theorem 1 - No Memory Leaks:

$$\text{NoLeaks}(S) \equiv \forall (hr\ n) \in \text{dom}(H) . \exists 1 \leq i \leq n . \exists x \in \text{dom}(\sigma_i) . (hr\ n) \in \text{TC}(H, \sigma_i(x))$$

(hr n) is reachable from the stack

6.2.2 Theorem 2 - No Dangling References

$$\begin{aligned} \text{NoDRInStack}_H(\sigma) &= \forall x \in \text{dom}(\sigma) . \sigma(x) \in \text{dom}(H) \cup \omega \\ \text{NoDRInHeap}(H) &= \forall (hr\ n) \in \text{dom}(H) . H((hr\ n)) \in \text{dom}(H) \cup \omega \\ \text{NoDR}(S) &\equiv \forall 1 \leq i \leq n . \text{NoDRInStack}_H(\sigma_i) \wedge \text{NoDRInHeap}(H) \end{aligned}$$

6.3 Lemma 1 - Single Owner and No Cycles

6.3.1 Statement of Lemma

We now prove a lemma which states that any memory allocated on the heap has exactly one owner heap reference pointing to it and that there are no cycles on the heap. Theorem 1 will follow as a corollary. First, some definitions:

$$\text{Let } S = \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle \quad \text{Where } S \in \text{State}$$

Def: $\text{owners} : (hr\ \omega) \rightarrow \mathcal{P}((hr\ \omega) \cup (\text{Loc}, \text{Frame}))$

$$\text{owners}_S((hr\ n)) = \{ (hr\ m) \mid H((hr\ m)) = (ohr\ n) \} \cup \{ (\sigma_i, x) \mid \sigma_i(x) = (ohr\ n) \} \quad \text{owners}((hr\ n)) \text{ is the set of everything that owns location } (hr\ n) \text{ in State } S$$

Now, the statement of the invariant to be proven:

$$\begin{aligned} \text{SingleOwnerNoCycle}(S) &\equiv (\forall (hr\ n) \in \text{dom}(H) . |\text{owners}_S((hr\ n))| = 1 \wedge (hr\ n) \notin \text{TC}(H, (hr\ n))) \wedge \\ &\quad \forall 2 \leq i \leq n . \sigma_{i-1}(\sigma_i.\text{var}) \notin (ohr\ \omega) \end{aligned}$$

In English, this means:

1. For every allocated cell, there is exactly one owner heap reference pointing to it in the stack or the heap.
2. There are no cycles of references.
3. This invariant is also strengthened to include that locations in the stack which are expecting to receive the return value for a function call in progress do not contain an owner heap reference

We prove this by induction on the execution of the operational semantics.

6.3.2 Proof of Lemma – Base Case

$$\text{InitialState} = \langle \text{Frame}(\{(x, 0)\}, \text{return } y, y) \mid \emptyset \mid x = \text{main}(); \text{return } x \rangle$$

Since the heap is empty and $n = 1$, the invariant is vacuously true in the base case.

6.3.3 Proof of Lemma – Inductive Case

On execution of commands that affect pointers - we assume inductively that all allocated cells on the heap have exactly one owner heap reference pointing to them before execution of a command. It should be self-evident from the definition of owners_S that only the creation, modification or destruction of owner heap reference values can change the number of owner heap references which point to a given allocated cell (changes to serf heap reference values and integer values can't affect the invariant). Similarly, by the definition of TC (which only follows owner heap references), it's clear that only the creation or change of owner heap reference values could create a cycle. So in conclusion, we need only concern ourselves with the impact of each command on owner heap reference values (and on the contents of previous stack frames, which by inspection are only modified during function call and return).

Our strategy will be to show that any allocated cell whose unique owner heap reference is affected by the command continues to have exactly one owner heap reference pointing to it after execution of the command.

Copy Assignment Rule 1:

$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad x \in \text{dom}(\sigma_n)}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x = e; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n[\text{val}/x] \mid H \mid c \rangle}$

By the typing rules, x cannot be an owner, since there is no typing rule which allows copy assignment to a variable which is an owner pointer. That is, we cannot write “ $x = \dots$ ”; if x is an owner pointer, we can only write “ $x := \dots$ ”(transfer assignment).

Therefore, we could not have lost an owner heap reference to something in the heap, since x can't contain an owner heap reference, nor could we gain an owner heap reference, since, again, by the typing rules, e does not have type τ owner $*$.

Copy Assignment Rule 2:

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr addr})}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v = e ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H [\text{val} / (\text{hr addr})] \mid c \rangle}$$

As in the above case, e cannot have type τ owner $*$, hence by type soundness, no owner heap references were lost or created in execution of this assignment.

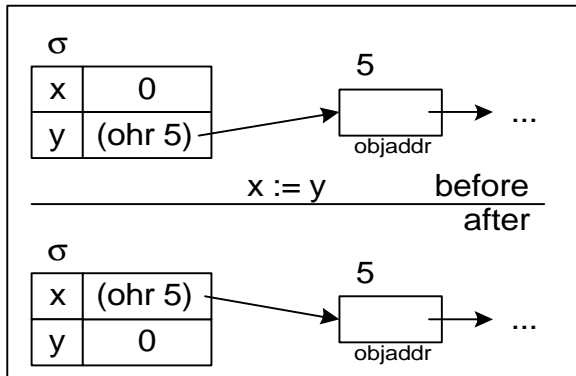
Copy Assignment Rule 3:

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{shr addr})}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v = e ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H [\text{val} / (\text{hr addr})] \mid c \rangle}$$

No owner heap references were destroyed or created, by the same argument as above.

Transfer Assignment Rule 1:

$$\frac{\begin{array}{l} x \in \text{dom}(\sigma_n) \quad y \in \text{dom}(\sigma_n) \\ \sigma_n(y) = (\text{ohr objaddr}) \\ \sigma_n(x) = 0 \end{array}}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := y ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [(\text{ohr objaddr}) / x] [0 / y] \mid H \mid c \rangle}$$



In the premises, we have $\sigma_n(x) = 0$, which means x points to null.

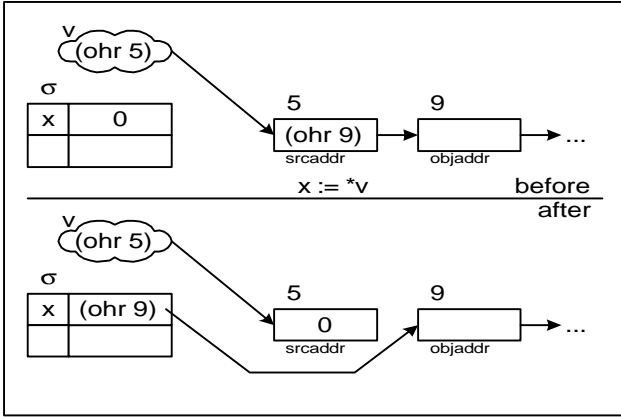
We also have $\sigma_n(y) = (\text{ohr objaddr})$, which means that y is an owner pointer to objaddr , and by assumption this is the only owner pointer to objaddr .

After we execute $\sigma_n[(\text{ohr objaddr})/x] [0/y]$, x becomes the owner pointer to objaddr and y points to null, thus objaddr still has exactly one owner pointer.

We assume there were no cycles before execution and none were created during execution, because the lack of address-of means a cycle can only be created by assigning a heap reference to an object on the heap (which by typing would have to be an owner heap reference).

Transfer Assignment Rule 2:

$$\frac{x \in \text{dom}(\sigma_n) \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr srcaddr}) \quad H(\text{hr srcaddr}) = (\text{ohr objaddr}) \quad \sigma_n(x) = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := *v ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n[(\text{ohr objaddr}) / x] \mid H[0 / (\text{hr srcaddr})] \mid c \rangle}$$



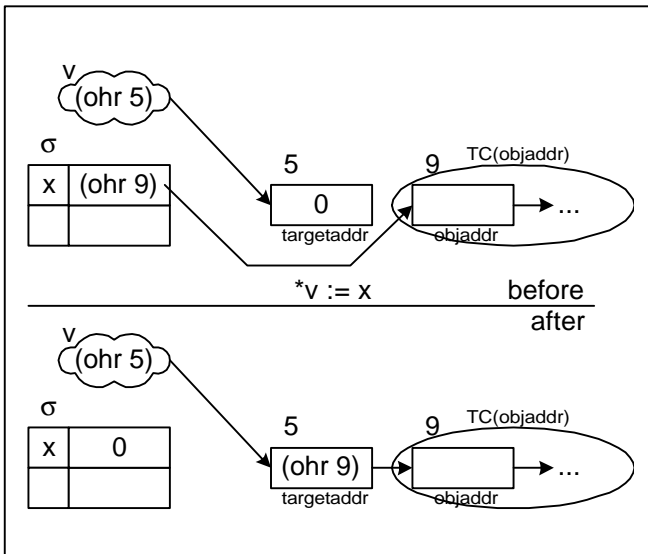
From the premises, we have that $\sigma_n(x) = 0$, which means x points to null, and that v evaluates to (ohr srcaddr) and $H(\text{hr srcaddr}) = (\text{ohr objaddr})$, so $*v$ is the only owner of (hr objaddr) , by assumption.

After we execute $\sigma_n[(\text{ohr objaddr}) / x]$, and $H[0 / (\text{hr srcaddr})]$, x becomes the sole owner of objaddr , and $*v$ points to null.

Again, we assume there were no cycles before execution and none were created during execution, since a cycle can only be created by assigning something on the heap.

Transfer Assignment Rule 3:

$$\frac{x \in \text{dom}(\sigma_n) \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr targetaddr}) \quad \sigma_n(x) = (\text{ohr objaddr}) \quad H(\text{hr targetaddr}) = 0 \quad (\text{hr targetaddr}) \notin \text{TC}(H, (\text{hr objaddr}))}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v := x ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n[0 / x] \mid H[(\text{ohr objaddr}) / (\text{hr targetaddr})] \mid c \rangle}$$



The premises give us that $\text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr targetaddr})$ and $H(\text{hr targetaddr}) = 0$, which means that $*v$ currently points to null. The premises also give us that $x \in \text{dom}(\sigma_n)$ and $\sigma_n(x) = (\text{ohr objaddr})$, thus x is an owner pointer to objaddr , and by assumption, it is the only owner pointer to objaddr .

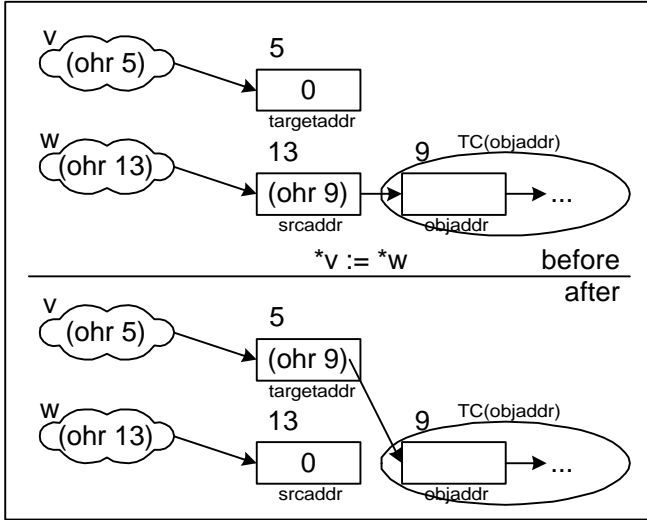
After we execute: $\sigma_n[0 / x]$ and $H[(\text{ohr objaddr}) / (\text{hr targetaddr})]$, $*v$ is the owner pointer to objaddr and x points to null. Thus, objaddr has a new owner pointer to it, but the old one has been destroyed, so there is again exactly one owner pointer to objaddr .

Remark: For two memory locations, $(\text{hr } n)$ and $(\text{hr } m)$ to be in the same cycle, it must be true that $(\text{hr } n)$ is in $\text{TC}((\text{hr } m))$ and $(\text{hr } m)$ is in $\text{TC}((\text{hr } n))$.

Before execution of the above there were no cycles, and we know by $(\text{hr targetaddr}) \notin \text{TC}(H, (\text{hr objaddr}))$ that targetaddr is not reachable from objaddr , hence, no cycles were created by making objaddr reachable from targetaddr , and therefore it remains true that there are no cycles.

Transfer Assignment Rule 4:

$$\frac{\begin{array}{c} \text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr targetaddr}) \quad \text{eval}(w, \sigma_n, H) \Downarrow (\text{ohr srcaddr}) \\ H(\text{hr srcaddr}) = (\text{ohr objaddr}) \\ H(\text{hr targetaddr}) = 0 \quad (\text{hr targetaddr}) \notin \text{TC}(H, (\text{ohr objaddr})) \end{array}}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v := *w ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H [(\text{ohr objaddr}) / (\text{hr targetaddr})] [0 / (\text{hr srcaddr})] \mid c \rangle}$$



Our premises give us that $\text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr targetaddr})$ and $H(\text{hr targetaddr}) = 0$, which means that $*v$ points to null. We also have that $\text{eval}(w, \sigma_n, H) \Downarrow (\text{ohr srcaddr})$ and $H(\text{hr srcaddr}) = (\text{ohr objaddr})$ which means that $*w$ is an owner of objaddr , and by assumption the only owner of objaddr .

When we execute $H [(\text{ohr objaddr}) / (\text{hr targetaddr})] [0 / (\text{hr srcaddr})]$, $*v$ now points to objaddr and $*w$ now points to null, so again objaddr has exactly one owner.

We began with no cycles, by assumption, and created no cycles, since $(\text{hr targetaddr}) \notin \text{TC}(H, (\text{ohr objaddr}))$, targetaddr is not reachable from objaddr so by making objaddr reachable from targetaddr , no cycles have been created, thus there are still no cycles.

Allocation Rule:

$$\frac{\sigma_n(x) = 0 \quad (\text{hr newaddr}) \notin \text{dom}(H)}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := \text{new} ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [(\text{ohr newaddr}) / x] \mid H \cup \{ ((\text{hr newaddr}), 0) \} \mid c \rangle}$$

By the premises, $\sigma_n(x) = 0$ means that x points to null, and $(\text{hr newaddr}) \notin \text{dom}(H)$ means that (hr newaddr) is an unallocated address and any owner pointers to it would be dangling. Technically, the `NoDanglingReferences` predicate should be strengthened to include the `SingleOwnerNoCycle` predicate, and the two proven simultaneously. However, it should be clear that merging the proofs is not problematic, and the possibility of a dangling self pointer invalidating the `SingleOwnerNoCycle` invariant is really a technical point, not a practical one.

After we execute $\sigma_n [(\text{ohr newaddr}) / x]$ and $H \cup \{ ((\text{hr newaddr}), 0) \}$, newaddr is owned by $\sigma_n(x)$ and therefore has exactly one owner. (Since nothing pointed to (hr newaddr) before, and now $\sigma_n(x)$ does, it has exactly one owner.)

By assumption, there were no cycles before the execution of $x := \text{new}$; since no pointer on the heap was re-assigned, no cycles were created, and it remains the case that there are no cycles.

Deallocation Rules:

$$\frac{\begin{array}{c} \sigma_n(x) = (\text{ohr addr}) \\ \forall y \in \text{dom}(\sigma_n) . y \neq x \wedge \sigma_n(y) = (\text{shr } z) \Rightarrow (\text{hr } z) \notin \text{TC}(H, (\text{ohr addr})) \end{array}}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{delete}(x) ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [0 / x] \mid \text{del}(H, (\text{ohr addr})) \mid c \rangle}$$

By the premises, $\sigma_n(x) = (\text{ohr addr})$ means that x is the owner pointer to addr , and by assumption, the only one.

Execution of $\sigma_n[0/x]$ means that (hr addr) no longer has an owner pointer. And $\text{del}(H, (\text{ohr addr}))$ means that everything in the transitive closure of (hr addr) has been removed, i.e., any owner pointers and the objects that they pointed to. (Owner pointers lost in deallocation point to deallocated objects, so their loss does not introduce objects with less than one owner.)

No cycles could have been created, since no pointer was assigned to any location on the heap.

$$\frac{\sigma_n(x) = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{delete}(x); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle}$$

The rule for this command is trivial and clearly does not affect the state of the stack or the heap, so the invariant is unchanged.

Function Call Rule (Copy & Transfer):

$$\frac{\begin{array}{l} f(x_1, \dots, x_m) \ x_{m+1}, \dots, x_p \ c_{\text{body}} \in F_{\text{op}} \\ \forall 1 \leq i \leq m. \text{eval}(e_i, \sigma_n, H) \Downarrow v_i \\ \forall m+1 \leq i \leq p. v_i = 0 \end{array}}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x = f(e_1, \dots, e_m); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n, \text{Frame}(\{(x_1, v_1)\dots, (x_p, v_p)\}, c, x) \mid H \mid c_{\text{body}} \rangle}$$

$$\frac{\begin{array}{l} f(x_1, \dots, x_m) \ x_{m+1}, \dots, x_p \ c_{\text{body}} \in F_{\text{op}} \\ \forall 1 \leq i \leq m. \text{eval}(e_i, \sigma_n, H) \Downarrow v_i \\ \forall m+1 \leq i \leq p. v_i = 0 \\ \sigma_n(x) = 0 \end{array}}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := f(e_1, \dots, e_m); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n, \text{Frame}(\{(x_1, v_1)\dots, (x_p, v_p)\}, c, x) \mid H \mid c_{\text{body}} \rangle}$$

For both these rules, the stack has been changed by the addition of a new frame, $\text{Frame}(\{(x_1, v_1)\dots, (x_p, v_p)\}, c, x)$, where the new VarValueMap contains each of the formal parameters and local variables, together with the objects to which they evaluate, v_1, \dots, v_m . Recall by typing that the formal parameters must have type int or $\tau \text{ serf}^*$, and therefore by soundness neither will contain owner heap references. All of the local variables are initialized to zero, therefore none of these may contain owner heap references. Therefore, because we have not created or destroyed any owner heap references, it's clear we haven't created any cycles or changed the number of owner pointers to any object, so the invariant holds.

In this rule, we create a new frame, so it's important to prove the third property of the invariant, i.e. $\forall 2 \leq i \leq n. \sigma_{i-1}(\sigma_i.\text{var}) \notin (\text{ohr } \omega)$. That is, we need to make sure the variable name we're placing in the new Frame.var field doesn't contain an owner heap reference in σ_n . By typing, the variable x in the first rule has type int , so by type soundness it can't possibly contain an owner heap reference. By the premise $\sigma_n(x) = 0$ in the second rule, we know $\sigma_n(x) \notin (\text{ohr } \omega)$.

Function Return Rule:

$$\frac{\forall y \in \text{dom}(\sigma_n). y \neq x \Rightarrow \sigma_n(y) \neq (\text{ohr } z)}{\langle \sigma_1, \dots, \sigma_{n-1}, \sigma_n \mid H \mid \text{return } x \rangle \rightarrow \langle \sigma_1, \dots, \sigma_{n-1}[\sigma_n(x) / \sigma_n.\text{var}] \mid H \mid \sigma_n.\text{cmd} \rangle}$$

From the premises of the **return** command, we have $\forall y \in \text{dom}(\sigma_n). y \neq x \Rightarrow \sigma_n(y) \neq (\text{ohr } z)$, which means that no variable in the current stack frame, other than x , could possibly contain an owner heap reference. Therefore, when we discard the σ_n stack frame, the only owner heap reference that could be affected is one contained in $\sigma_n(x)$. (If $\sigma_n(x)$ is not an owner heap reference, then the command affects no owner heap references and clearly maintains the invariant).

If $\sigma_n(x)$ does contain an owner heap reference, then by assumption it is the sole owner of the memory to which it points. When we execute $\sigma_{n-1}[\sigma_n(x) / \sigma_n.\text{var}]$, the variable $\sigma_{n-1}(\sigma_n.\text{var})$, which exists in the $n-1^{\text{th}}$ stack frame, receives the owner heap reference. By the third clause of the invariant, we know this action does not overwrite any owner heap reference. After the execution of **return** x , the function call ends and σ_n disappears, taking x with it, leaving the newly assigned variable as the sole owner of the memory to which it points.

No element of the heap was reassigned, so no cycles were created. By assumption, there were no cycles before execution of the command, and hence none after execution of the command.

Skip, While, and Conditional Rule:

$\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{skip}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle$
$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} \neq 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{while } e \text{ do } c_{\text{loop}}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c_{\text{loop}}; \text{while } e \text{ do } c_{\text{loop}}; c \rangle}$
$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{while } e \text{ do } c_{\text{loop}}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle}$
$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} \neq 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{if } e \text{ then } c_1 \text{ else } c_2; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c_1; c \rangle}$
$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{if } e \text{ then } c_1 \text{ else } c_2; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c_2; c \rangle}$

By assumption, before we execute the command there is exactly one owner pointer to each allocated object on the heap and there are no cycles. None of these commands modify the stack or the heap in any way, so clearly the invariant must still hold after executing any of them.

6.3.4 Proof of Lemma – Conclusion

Therefore, by induction on the execution of the operational semantics, the SingleOwnerNoCycle(S) invariant holds throughout the entire execution of any legal Feudal C program.

6.4 Theorem 1 - No Memory Leaks:

6.4.1 Statement of Theorem

A memory leak is allocated memory that is not reachable from the stack, i.e., is not in the transitive closure of anything on the stack. Feudal C is safe with respect to memory leaks.

$\text{NoLeaks}(S) \equiv \forall (hr\ n) \in \text{dom}(H) . \exists 1 \leq i \leq n . \exists x \in \text{dom}(\sigma_i) . (hr\ n) \in \text{TC}(H, \sigma_i(x))$

(hr n) is reachable from the stack

6.4.2 Proof of Theorem

By lemma 1, the SingleOwnerNoCycle(S) invariant holds throughout the entire execution of any legal Feudal C program. This directly implies the NoLeaks() invariant also holds at each of these states, by the following argument:

For any state S during the execution of a legal Feudal C program, Lemma 1 tells us that every allocated cell, (hr n), on the heap has exactly one owner and there are no cycles. Choose any arbitrary $(hr\ n) \in \text{dom}(H)$. Because every object in the heap has exactly one owner, we can follow a simple path of owner pointers back from (hr n) (at each step moving to the location of the current cell's unique owner heap reference). We have either a finite or an infinite list of predecessors. If the list is infinite we either have a cycle (which is impossible, by Lemma 1), or an infinite list of allocated memory, (which is impossible because the heap is initially empty and at any state in the program's execution we could only have executed a finite number of allocations). Therefore we must have a finite list. If the finite list of predecessors ends in the heap, this would imply there exists a memory location with no owner pointer (which is also impossible, by Lemma 1). Therefore, the list of predecessors must end on the stack, and hence (hr n) is reachable from the stack (exists in the transitive closure of $\sigma_i(x)$ for some i and $x \in \text{dom}(\sigma_i)$).

6.5 Lemma 2 – No serfs pointing down

6.5.1 Statement of Lemma

Let $S = \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle$.

The invariant to be proven is:

$\text{NoSerfsDown}(S) \equiv \forall 2 \leq j \leq n. \forall 1 \leq i \leq j-1. \forall x \in \text{dom}(\sigma_i). \forall (y, (\text{ohr addr})) \in \sigma_j.\text{vvm}. \sigma_i(x) \notin \text{TC}(H, (\text{hr addr}))$

We prove this by induction on the execution of the operational semantics.

6.5.2 Proof of Lemma – Base Case

$\text{InitialState} = \langle \text{Frame}(\{(x, 0)\}), \text{return } y, y \mid \emptyset \mid x = \text{main}(); \text{return } x \rangle$

The invariant holds vacuously because $\neg \exists j \in [2, n]$.

6.5.3 Proof of Lemma – Inductive Case

Function Call (Copy) Rule:

$$\frac{\begin{array}{c} f(x_1, \dots, x_m) \ x_{m+1}, \dots, x_p \ c_{\text{body}} \in F_{\text{op}} \\ \forall 1 \leq i \leq m. \text{eval}(e_i, \sigma_n, H) \Downarrow v_i \\ \forall m+1 \leq i \leq p. v_i = 0 \end{array}}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x = f(e_1, \dots, e_m); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n, \text{Frame}(\{(x_1, v_1)\dots, (x_p, v_p)\}), c, x \mid H \mid c_{\text{body}} \rangle}$$

By the typing rules, none of the e_1, \dots, e_m are of type τ owner*. The local variables all have the value 0. Therefore there are no (ohr addr) values in the current stack frame ($j=n$). Inductively, the property holds for $j < n$, so it holds $\forall j$.

Function Call (Transfer) Rule:

$$\frac{\begin{array}{c} f(x_1, \dots, x_m) \ x_{m+1}, \dots, x_p \ c_{\text{body}} \in F_{\text{op}} \\ \forall 1 \leq i \leq m. \text{eval}(e_i, \sigma_n, H) \Downarrow v_i \\ \forall m+1 \leq i \leq p. v_i = 0 \\ \sigma_n(x) = 0 \end{array}}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := f(e_1, \dots, e_m); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n, \text{Frame}(\{(x_1, v_1)\dots, (x_p, v_p)\}), c, x \mid H \mid c_{\text{body}} \rangle}$$

Again, no local owner pointers exist.

Function Return Rule:

$$\frac{\forall y \in \text{dom}(\sigma_n). y \neq x \Rightarrow \sigma_n(y) \neq (\text{ohr } z)}{\langle \sigma_1, \dots, \sigma_{n-1}, \sigma_n \mid H \mid \text{return } x \rangle \rightarrow \langle \sigma_1, \dots, \sigma_{n-1}[\sigma_n(x) / \sigma_n.\text{var}] \mid H \mid \sigma_n.\text{cmd} \rangle}$$

Inductively, the invariant holds for $j \in [2, n-2]$. For $j = n-1$, we have changed the value of the return variable. If the return value (val) was an integer, no pointer values changed. If the return value was an owner pointer, by the inductive hypothesis, no serfs in higher frames pointed to the object (or any of its descendants). Any serfs in σ_n have disappeared with the frame. Therefore no serfs point to the object or its descendants, and the property holds.

Allocation Rule:

$$\frac{\sigma_n(x) = 0 \quad (\text{hr newaddr}) \notin \text{dom}(H)}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := \text{new}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n[(\text{ohr newaddr})/x] \mid H \cup \{((\text{hr newaddr}), 0)\} \mid c \rangle}$$

A new owner pointer has been created in the local frame. Only a dangling reference from a previous frame could create a serf pointer to the new object. Technically, the NoDanglingReferences predicate should be strengthened to include the NoSerfsDown predicate, and the two proven simultaneously. However, it should be clear that merging the proofs is not problematic, and the possibility of a dangling serf pointer invalidating the NoSerfsDown invariant is really a technical point, not a practical one.

Deallocation Rules:

$$\frac{\frac{\sigma_n(x) = (\text{ohr addr})}{\forall y \in \text{dom}(\sigma_n) . y \neq x \wedge \sigma_n(y) = (\text{shr } z) \Rightarrow (\text{hr } z) \notin \text{TC}(H, (\text{hr addr}))}}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{delete}(x); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n[0/x] \mid \text{del}(H, (\text{ohr addr})) \mid c \rangle}}{\sigma_n(x) = 0}$$

$$\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{delete}(x); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle$$

No serf pointers have been created or changed, and there are no new objects. The invariant holds.

Skip, While & Conditional Rules:

$$\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{skip}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle$$

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} \neq 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{while } e \text{ do } c_{\text{loop}}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c_{\text{loop}}; \text{while } e \text{ do } c_{\text{loop}}; c \rangle}}$$

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{while } e \text{ do } c_{\text{loop}}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle}}$$

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} \neq 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{if } e \text{ then } c_1 \text{ else } c_2; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c_1; c \rangle}}$$

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{if } e \text{ then } c_1 \text{ else } c_2; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c_2; c \rangle}}$$

Clearly none of these commands modifies the stack or the heap, (in particular, recall expressions do not have side effects) so the invariant holds.

Copy Assignment Rules:

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad x \in \text{dom}(\sigma_n)}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x = e; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n[\text{val}/x] \mid H \mid c \rangle}}$$

By the typing rules for copy assignment, the type of x must be either an int or τ serf*. If it is an integer, it does not affect the property. Otherwise, it is a serf pointer. Serf pointers in the local frame (σ_n), such as x , are unrestricted by the property. Therefore it

continues to hold.

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr addr})}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v = e ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H [\text{val} / (\text{hr addr})] \mid c \rangle}$$

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{shr addr})}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v = e ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H [\text{val} / (\text{hr addr})] \mid c \rangle}$$

By the typing rules, the type of e and $*v$ must be `int`. Therefore, the serf and owner pointer structure is unaffected.

Transfer Assignment Rules:

$$\frac{x \in \text{dom}(\sigma_n) \quad y \in \text{dom}(\sigma_n) \quad \sigma_n(y) = (\text{ohr objaddr}) \quad \sigma_n(x) = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := y ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [(\text{ohr objaddr}) / x] [0 / y] \mid H \mid c \rangle}$$

This simply transfers ownership within the local stack frame. This does not affect serfs in higher frames.

$$\frac{x \in \text{dom}(\sigma_n) \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr srcaddr}) \quad H(\text{hr srcaddr}) = (\text{ohr objaddr}) \quad \sigma_n(x) = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := *v ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [(\text{ohr objaddr}) / x] \mid H [0 / (\text{hr srcaddr})] \mid c \rangle}$$

$$\frac{x \in \text{dom}(\sigma_n) \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr targetaddr}) \quad \sigma_n(x) = (\text{ohr objaddr}) \quad H((\text{hr targetaddr})) = 0 \quad (\text{hr targetaddr}) \notin \text{TC}(H, (\text{ohr objaddr}))}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v := x ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [0 / x] \mid H [(\text{ohr objaddr}) / (\text{hr targetaddr})] \mid c \rangle}$$

$$\frac{\text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr targetaddr}) \quad \text{eval}(w, \sigma_n, H) \Downarrow (\text{ohr srcaddr}) \quad H(\text{hr srcaddr}) = (\text{ohr objaddr}) \quad H((\text{hr targetaddr})) = 0 \quad (\text{hr targetaddr}) \notin \text{TC}(H, (\text{ohr objaddr}))}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v := *w ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H [(\text{ohr objaddr}) / (\text{hr targetaddr})] [0 / (\text{hr srcaddr})] \mid c \rangle}$$

In each case, we are rearranging the ownership structure of objects reachable from the current stack frame; but the objects are still all reachable from the current stack frame. Therefore, if there were no serfs pointing to these objects before, there still aren't, and therefore the invariant holds.

6.5.4 Conclusion

Therefore, by induction on the execution of the operational semantics, the `NoSerfsPointingDown(S)` invariant holds throughout the entire execution of any legal Feudal C program.

6.6 Lemma 3 – No serfs on heap

By the typing rules, no type allows serf pointers to exist on the heap. Therefore, by type soundness of the operational semantics, there will never be a serf heap reference stored in the heap at any state.

6.7 Theorem 2 - No Dangling References

6.7.1 Statement of Theorem

We now prove a theorem which states that any serf or owner heap references in the stack or heap always point to an allocated heap address. First, some definitions:

Let $S = \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle$.

$\text{NoDRInStack}_H(\sigma) = \forall x \in \text{dom}(\sigma) . \sigma(x) \in \text{dom}(H) \cup \omega$

$\text{NoDRInHeap}(H) = \forall (\text{hr } n) \in \text{dom}(H) . H(\text{hr } n) \in \text{dom}(H) \cup \omega$

Now, the statement of the invariant to be proven:

$\text{NoDR}(S) \equiv \forall 1 \leq i \leq n . \text{NoDRInStack}_H(\sigma_i) \wedge \text{NoDRInHeap}(H)$

We prove this by induction on the execution of the operational semantics.

6.7.2 Proof of Evaluation Results Mini-Lemma

We begin by showing that the only values which may result from executing $\text{eval}(e, \sigma_n, H) \Downarrow \text{val}$ are integers, or serf/owner heap references which already exist in the stack or on the heap. Define:

$\text{PossibleEvalReturns}(S) = \{ \text{val} \mid \exists e \in \text{AExpr} . \text{eval}(e, \sigma_n, H) \Downarrow \text{val} \}$

$\text{InStackOrHeap}(S) = \{ \text{val} \mid \exists 1 \leq i \leq n . \exists x \in \text{dom}(\sigma_i) . \sigma_i(x) = \text{val} \} \cup$
 $\{ (\text{ohr } n) \mid \exists (\text{hr } m) \in \text{dom}(H) . H(\text{hr } m) = (\text{ohr } n) \} \cup$
 $\{ (\text{shr } n) \mid \exists (\text{hr } m) \in \text{dom}(H) . H(\text{hr } m) = (\text{ohr } n) \} \cup$
 ω

Now, we show : $\text{PossibleEvalReturns}(S) \subseteq \text{InStackOrHeap}(S)$

The proof is by structural induction on the $\text{eval}()$ derivation.

Integer values:

$$\frac{n \in \omega}{\text{eval}(n, \sigma, H) \Downarrow n}$$

Because $n \in \omega$, we clearly have $n \in \text{InStackOrHeap}(S)$

Arithmetic operations:

$$\frac{\text{eval}(e_1, \sigma, H) \Downarrow n_1 \quad n_1 \in \omega \quad \text{eval}(e_2, \sigma, H) \Downarrow n_2 \quad n_2 \in \omega \quad n \text{ is } n_1 \text{ op } n_2}{\text{eval}(e_1 \text{ op } e_2, \sigma, H) \Downarrow n}$$

Because $n_1 \in \omega$ and $n_2 \in \omega$, by arithmetic we know $n \in \omega$, and hence $n \in \text{InStackOrHeap}(S)$.

Stack Variable Lookup Rule:

$$\frac{x \in \text{dom}(\sigma)}{\text{eval}(x, \sigma, H) \Downarrow \sigma(x)}$$

Our premises state that $x \in \text{dom}(\sigma)$, and clearly $\sigma(x)$ is a value on the stack. Therefore $\text{eval}(x, \sigma, H)$ does in fact evaluate to something that is already on the stack ($\sigma(x) \in \text{InStackOrHeap}(S)$).

Heap Lookup Rules:

$$\frac{\text{eval}(v, \sigma, H) \Downarrow (\text{ohr addr}) \quad \text{val} = H(\text{hr addr})}{\text{eval}(*v, \sigma, H) \Downarrow \text{val}}$$

By the inductive hypothesis on the derivation of premise $\text{eval}(v, \sigma, H) \Downarrow (\text{ohr addr})$, we know $(\text{ohr addr}) \in \text{InStackOrHeap}(S)$.

By the premise $\text{val} = H(\text{hr addr})$, we know val is stored at (hr addr) , and hence is on the heap. Thus $\text{eval}(*v, \sigma, H) \Downarrow \text{val}$ does result in a value that is on the heap ($\text{val} \in \text{InStackOrHeap}(S)$).

$$\frac{\text{eval}(v, \sigma, H) \Downarrow (\text{shr addr}) \quad \text{val} = H(\text{hr addr}) \quad \text{val} \notin (\text{ohr } \omega)}{\text{eval}(*v, \sigma, H) \Downarrow \text{val}}$$

By the same reasoning as the previous rule, val is a value on the heap, so $\text{val} \in \text{InStackOrHeap}(S)$.

$$\frac{\text{eval}(v, \sigma, H) \Downarrow (\text{shr addr}) \quad (\text{ohr } x) = H(\text{hr addr})}{\text{eval}(*v, \sigma, H) \Downarrow (\text{shr } x)}$$

By the inductive hypothesis on the derivation of premise $\text{eval}(v, \sigma, H) \Downarrow (\text{shr addr})$, we know $(\text{shr addr}) \in \text{InStackOrHeap}(S)$. Now, by the premise $(\text{ohr } x) = H(\text{hr addr})$ and the third clause of the definition of $\text{InStackOrHeap}(S)$, we know $(\text{shr } x) \in \text{InStackOrHeap}(S)$.

6.7.3 Proof of Theorem – Base Case

$\text{InitialState} = \langle \text{Frame}(\{(x, 0)\}, \text{return } y, y) \mid \emptyset \mid x = \text{main}(); \text{return } x \rangle$

Since there are no serf or owner heap references in the stack or heap, the lemma is vacuously true in the base case.

6.7.4 Proof of Theorem – Inductive Case

Function Call (Copy) Rule:

$$\frac{\begin{array}{l} f(x_1, \dots, x_m) \quad x_{m+1}, \dots, x_p \quad c_{\text{body}} \in F_{\text{op}} \\ \forall 1 \leq i \leq m. \text{eval}(e_i, \sigma_n, H) \Downarrow v_i \\ \forall m+1 \leq i \leq p. v_i = 0 \end{array}}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x = f(e_1, \dots, e_m); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n, \text{Frame}(\{(x_1, v_1)\dots, (x_p, v_p)\}, c, x) \mid H \mid c_{\text{body}} \rangle}$$

The stack has been changed by the addition of a new frame, $\text{Frame}(\{(x_1, v_1)\dots, (x_p, v_p)\}, c, x)$, which contains local variables, and the objects to which they evaluate, v_1, \dots, v_m , which by the mini-lemma are values that already exist on the stack in previous frames or on the heap. By inductive assumption, they are not dangling references, and will not be so, when they are put on the new frame. Hence, no dangling references have been added.

Function Call (Transfer) Rule:

$$\frac{\begin{array}{l} f(x_1, \dots, x_m) \quad x_{m+1}, \dots, x_p \quad c_{\text{body}} \in F_{\text{op}} \\ \forall 1 \leq i \leq m. \text{eval}(e_i, \sigma_n, H) \Downarrow v_i \\ \forall m+1 \leq i \leq p. v_i = 0 \\ \sigma_n(x) = 0 \end{array}}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := f(e_1, \dots, e_m); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n, \text{Frame}(\{(x_1, v_1)\dots, (x_p, v_p)\}, c, x) \mid H \mid c_{\text{body}} \rangle}$$

By the same argument as above, no new dangling references have been created.

Function Return Rule:

$$\frac{\forall y \in \text{dom}(\sigma_n) . y \neq x \Rightarrow \sigma_n(y) \neq (\text{ohr } z)}{\langle \sigma_1, \dots, \sigma_{n-1}, \sigma_n \mid H \mid \text{return } x \rangle \rightarrow \langle \sigma_1, \dots, \sigma_{n-1} [\sigma_n(x) / \sigma_n.\text{var}] \mid H \mid \sigma_n.\text{cmd} \rangle}$$

By assumption, $\sigma_n(x)$ is not a dangling pointer, so when the variable $\sigma_n.\text{var}$ in σ_{n-1} is assigned to $\sigma_n(x)$, it will also not be a dangling pointer. All pointers in the stack frame, σ_n , will cease to exist after the stack frame is popped, and hence there will be no dangling pointers created.

Allocation Rule:

$$\frac{\sigma_n(x) = 0 \quad (\text{hr newaddr}) \notin \text{dom}(H)}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := \text{new} ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [(\text{ohr newaddr}) / x] \mid H \cup \{((\text{hr newaddr}), 0)\} \mid c \rangle}$$

We are putting an address that is not in the heap in the location designated by x , and simultaneously adding it to the heap, giving it the value zero. Therefore, x is pointing to something that is on the heap, and $H((\text{hr newaddr}))$ has value zero, thus neither of these is a dangling reference. Thus the invariant holds.

Deallocation Rules:

$$\frac{\begin{array}{c} \sigma_n(x) = (\text{ohr addr}) \\ \forall y \in \text{dom}(\sigma_n) . y \neq x \wedge \sigma_n(y) = (\text{shr } z) \Rightarrow (\text{hr } z) \notin \text{TC}(H, (\text{hr addr})) \end{array}}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{delete}(x); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [0 / x] \mid \text{del}(H, (\text{ohr addr})) \mid c \rangle}$$

$$\frac{\sigma_n(x) = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{delete}(x); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle}$$

By the inductive assumption, all references existing before the execution of `delete` were not dangling. Execution of `delete` reassigns x so that it no longer points to something on the heap, and now points to null. A pointer to null is not considered to be dangling. The references in the transitive closure of (hr addr) are all deallocated by `del()`, and hence clearly not dangling. By the statement, $\forall y \in \text{dom}(\sigma_n) . y \neq x \wedge \sigma_n(y) = (\text{shr } z) \Rightarrow (\text{hr } z) \notin \text{TC}(H, (\text{hr addr}))$, in the premises, nothing in σ_n pointed to anything in $\text{TC}(H, (\text{hr addr}))$. Furthermore, no serfs in $\sigma_1, \dots, \sigma_{n-1}$ point to $\text{TC}(H, (\text{hr addr}))$ by Lemma 2, and no other owners may point to anything in $\text{TC}(H, (\text{hr addr}))$, by Lemma 1.

In summary, the only changes have been to remove pointers and hence, since all other pointers were not dangling before the deletions, they are not dangling afterward.

In the second rule, there is no change to the contents of the heap or stack, so the invariant holds.

Skip, While & Conditional Rules:

$\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{skip}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle$
$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} \neq 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{while } e \text{ do } c_{\text{loop}}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c_{\text{loop}}; \text{while } e \text{ do } c_{\text{loop}}; c \rangle}$
$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{while } e \text{ do } c_{\text{loop}}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle}$
$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} \neq 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{if } e \text{ then } c_1 \text{ else } c_2; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c_1; c \rangle}$
$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{if } e \text{ then } c_1 \text{ else } c_2; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c_2; c \rangle}$

Clearly none of these commands modifies the stack or the heap, (in particular, recall expressions do not have side effects) so the invariant holds.

Copy Assignment Rules:

$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad x \in \text{dom}(\sigma_n)}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x = e; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [\text{val} / x] \mid H \mid c \rangle}$
--

After the execution of this command, x contains an integer or a pointer which was evaluated in the context of σ_n . If it's an integer, then by typing and soundness the command doesn't affect any heap references and the invariant holds. If it's a heap reference, then by the mini-lemma, it must be a heap reference that already exists on the stack or the heap, therefore by the inductive hypothesis this value cannot be a dangling reference.

$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr addr})}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v = e; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H [\text{val} / (\text{hr addr})] \mid c \rangle}$
$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{shr addr})}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v = e; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H [\text{val} / (\text{hr addr})] \mid c \rangle}$

By typing, e and *v must have type int, so by type soundness no owner heap references are affected by the command and the invariant holds.

Transfer Assignment Rules:

$\frac{x \in \text{dom}(\sigma_n) \quad y \in \text{dom}(\sigma_n) \quad \sigma_n(y) = (\text{ohr objaddr}) \quad \sigma_n(x) = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := y; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [(\text{ohr objaddr}) / x] [0 / y] \mid H \mid c \rangle}$

By assumption, $\sigma_n(y)$ is not a dangling reference. We assign x the location pointed to by y and nothing is done to that location, thus $\sigma_n(x)$ is not a dangling reference after reassignment. The value of $\sigma_n(y)$ is set to null, and hence is not a dangling reference. Thus the invariant holds.

$$\frac{
\begin{array}{l}
x \in \text{dom}(\sigma_n) \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr srcaddr}) \\
H(\text{hr srcaddr}) = (\text{ohr objaddr}) \\
\sigma_n(x) = 0
\end{array}
}{
\langle \sigma_1, \dots, \sigma_n \mid H \mid x := *v ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [(\text{ohr objaddr}) / x] \mid H [0 / (\text{hr srcaddr})] \mid c \rangle
}$$

By the premises and the mini-lemma, both (hr srcaddr) and (hr objaddr) are valid addresses on the heap. After we execute the transfer assignment, x is pointing to (ohr objaddr), which is still valid, and H(hr srcaddr) is pointing to null, hence neither reassignment has resulted in a dangling reference, and the invariant holds.

$$\frac{
\begin{array}{l}
x \in \text{dom}(\sigma_n) \quad \sigma_n(x) = (\text{ohr objaddr}) \\
\text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr targetaddr}) \\
H((\text{hr targetaddr})) = 0 \quad (\text{hr targetaddr}) \notin \text{TC}(H, (\text{hr objaddr}))
\end{array}
}{
\langle \sigma_1, \dots, \sigma_n \mid H \mid *v := x ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [0 / x] \mid H [(\text{ohr objaddr}) / (\text{hr targetaddr})] \mid c \rangle
}$$

$$\frac{
\begin{array}{l}
\text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr targetaddr}) \quad \text{eval}(w, \sigma_n, H) \Downarrow (\text{ohr srcaddr}) \\
H(\text{hr srcaddr}) = (\text{ohr objaddr}) \\
H((\text{hr targetaddr})) = 0 \quad (\text{hr targetaddr}) \notin \text{TC}(H, (\text{hr objaddr}))
\end{array}
}{
\langle \sigma_1, \dots, \sigma_n \mid H \mid *v := *w ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H [(\text{ohr objaddr}) / (\text{hr targetaddr})] [0 / (\text{hr srcaddr})] \mid c \rangle
}$$

By the inductive hypothesis, (ohr objaddr) is a valid heap reference, which we assign to H(hr targetaddr) and thus H(hr targetaddr) is not a dangling reference. Since $\sigma_n(x)$ and H(hr srcaddr) are assigned 0, they are not dangling, therefore the invariant holds.

6.7.5 Conclusion

Therefore, by induction on the execution of the operational semantics, the NoDR(S) invariant holds throughout the entire execution of any legal Feudal C program.

6.8 Theorem of Memory Safety

By Theorem 1 and 2 we know that the invariants NoLeaks(S) and NoDR(S) hold for all states throughout the entire execution of any legal Feudal C program. Therefore, by the definition of memory safety, Feudal C is memory safe.

7 Future Work

Due to time constraints, we have not proven type soundness in this paper. Future work includes such a proof.

The operational semantics checks many facts about the runtime state (e.g., $\sigma_n(x)=0$ in the allocation rule). These checks are primarily the mechanism by which we ensure the preprocessor has done its job correctly. If we formalize the preprocessor, and prove that a preprocessed program always makes progress (modulo certain inevitabilities like null dereference), we can remove the checks (thereby justifying a claim of no runtime overhead).

The language and type system, as they stand, are probably not suitable for general-purpose programming. Certain language features, especially recursive types, need to be added. Moreover, the type system should be extended, as outlined in Appendix D. We plan to make these improvements while preserving the memory safety of the system.

8 Conclusion

Feudal C is the first step towards automatic memory management in the type system. The language and type system presented here perform all memory management safely, requiring only minimal runtime overhead (with the assistance of the preprocessor). However, in this form, they are too restricted for general-purpose use; this is the subject of future work.

9 Bibliography

- [Ca96] Cardelli, Luca. “Type Systems”, CRC Handbook of Computer Science and Engineering, Ch. 140, September, 1996.
- [Ch92] Chirimar, J.; Gunter, C.A.; Riecke, J.G. “Proving Memory Management Invariants for a Language Based on Linear Logic”, ACM, 1992. p.139-50.
- [Er93] Eyre-Todd, R.A. “The detection of dangling references in C++ programs.”, ACM Letters on Programming Languages and Systems, vol.2, (no.1-4), March-Dec. 1993. p.127-34.
- [LC99] LCLint v2.4 Homepage and Documentation. <http://www.sds.lcs.mit.edu/lclint/>
- [Ne99] Necula, George. “CS263 Course Lecture Notes”, Spring 1999. http://www-nt.cs.berkeley.edu/home/necula/public_html/cs263
- [Wi95] Wilson, Paul R., Mark S. Johnstone, Michael Neely, and David Boles. “Dynamic Storage Allocation: A Survey and Critical Review. In International Workshop on Memory Management”, Kinross, Scotland, UK, September 1995.
- [Wi96] Winskel, Glynn. “The Formal Semantics of Programming Languages – An Introduction”, MIT 1996.

Appendix A – Complete Syntax for Feudal C

Implicitly pre-defined sets:

ω - The set of all non-negative integer literals, i.e. $\{0, 1, \dots\}$

Loc – The set of all legal variable names

Fname – The set of all legal function names (defined s.t. $\text{Loc} \cap \text{Fname} = \emptyset$)

<p>Aexpr ::=</p> <ul style="list-style-type: none"> n $n \in \omega$ v $v \in \text{Lvalexpr}$ e₁ op e₂ $e_1, e_2 \in \text{Aexpr}, \text{op} \in \{+, *, \&\&, , ==, !=\}$ <p>TypeAux ::=</p> <ul style="list-style-type: none"> int t owner * $t \in \text{TypeAux}$ <p>Type ::=</p> <ul style="list-style-type: none"> t $t \in \text{TypeAux}$ t serf * $t \in \text{TypeAux}$ <p>DeclSeq ::=</p> <ul style="list-style-type: none"> t x ; ds $t \in \text{Type}, x \in \text{Loc}, ds \in \text{DeclSeq}$ ϵ <p>GdeclSeq ::=</p> <ul style="list-style-type: none"> g₁ ; g₂ $g_1, g_2 \in \text{GdeclSeq}$ t f(ds₁) ds₂ c ; rc $t \in \text{Type}, f \in \text{Fname}, ds_1, ds_2 \in \text{DeclSeq},$ $c \in \text{Com}, rc \in \text{RetCom}$ <p>Lvalexpr ::=</p> <ul style="list-style-type: none"> x $x \in \text{Loc}$ *v $v \in \text{Lvalexpr}$ <p>RetCom ::=</p> <ul style="list-style-type: none"> return x $x \in \text{Loc}$ <p>Com ::=</p> <ul style="list-style-type: none"> c₁ ; c₂ $c_1, c_2 \in \text{Com}$ if e then c₁ else c₂ $c_1, c_2 \in \text{Com}, e \in \text{Aexpr}$ while e do c₁ $c_1 \in \text{Com}, e \in \text{Aexpr}$ skip v = e $v \in \text{Lvalexpr}, e \in \text{Aexpr}$ v₁ := v₂ $v_1, v_2 \in \text{Lvalexpr}$ x = f(e₁, ..., e_n) $x \in \text{Loc}, f \in \text{Fname}, e_1, \dots, e_n \in \text{Aexpr}$ x := f(e₁, ..., e_n) $x \in \text{Loc}, f \in \text{Fname}, e_1, \dots, e_n \in \text{Aexpr}$ x := new(t) $x \in \text{Loc}, t \in \text{Type}$ delete(x) $x \in \text{Loc}$ 	<p>Arithmetic Expressions</p> <p><i>Numeric literals</i></p> <p><i>Variable reference</i></p> <p><i>Binary Arithmetic operator</i></p> <p>Types that may exist in the heap</p> <p><i>Basic integer type</i></p> <p><i>Owner pointer</i></p> <p>Types</p> <p><i>Integers and owner pointers</i></p> <p><i>Serf pointer</i></p> <p>Declaration Sequence (zero or more)</p> <p><i>One or more Declarations</i></p> <p><i>Zero Declarations</i></p> <p>Declarations that can be made at the global scope</p> <p><i>Global declaration sequencing</i></p> <p><i>Function Definition</i></p> <p>L-valued variable expressions</p> <p><i>Simple Variable</i></p> <p><i>Dereferenced pointer</i></p> <p>Auxilliary Command Rule</p> <p><i>Function return</i></p> <p>Commands</p> <p><i>Command sequencing</i></p> <p><i>Conditional</i></p> <p><i>Looping</i></p> <p><i>NOP</i></p> <p><i>Assignment - copy</i></p> <p><i>Assignment - transfer</i></p> <p><i>Function call - copy</i></p> <p><i>Function call - transfer</i></p> <p><i>Dynamic allocation</i></p> <p><i>Dynamic de-allocation</i></p>
--	---

Notes:

A GdeclSeq is a syntactically legal program.

Appendix B – Complete Typing Rules for Feudal C

Judgements

$\vdash F$	F is a well-formed function environment (program)
$\vdash^F f$	f is a well-typed function
$\Gamma \vdash^F c$	Command c is well-typed given type environment Γ and function environment F
$\Gamma \vdash rc : \tau$	Return command rc returns type τ , given type environment Γ
$\Gamma \vdash e : \tau$	Expression e has type τ in type environment Γ

Integer Literals

$$\frac{n \in \omega}{\Gamma \vdash n : \text{int}}$$

Local Variable Reference

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Pointer Dereference

$$\frac{\Gamma \vdash v : \tau \text{ owner } *}{\Gamma \vdash *v : \tau}$$

$$\frac{\Gamma \vdash v : \text{int serf } *}{\Gamma \vdash *v : \text{int}}$$

$$\frac{\Gamma \vdash v : \tau \text{ owner } * \text{ serf } *}{\Gamma \vdash *v : \tau \text{ serf } *}$$

Arithmetic Expressions

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \text{op} \in \{+, *, \&\&, ||, ==, !=\}}{\Gamma \vdash e_1 \text{ op } e_2 : \text{int}}$$

Program

$$\frac{\forall f \in F. \vdash^F f}{\vdash F}$$

Function Definition

$$\frac{\begin{array}{l} \tau f (x_1 : \tau_1, \dots, x_n : \tau_n) x_{n+1} : \tau_{n+1}, \dots, x_m : \tau_m c ; rc \in F \\ \Gamma = x_1 : \tau_1, \dots, x_m : \tau_m \\ \Gamma \vdash^F c \\ \Gamma \vdash rc : \tau \\ \tau \neq \tau' \text{ serf } * \\ \forall 1 \leq i \leq n. \tau_i \neq \tau'_i \text{ owner } * \end{array}}{\vdash^F f}$$

Function Return Command

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash \text{return } x : \tau}$$

Function Call - Copy

$$\frac{\begin{array}{c} \tau f (x_1 : \tau_1, \dots, x_n : \tau_n) x_{n+1} : \tau_{n+1}, \dots, x_m : \tau_m c ; rc \in F \\ \forall 1 \leq i \leq n. \Gamma \vdash e_i : \tau_i \\ \Gamma \vdash x : \tau \\ \tau \neq \tau' \text{ owner}^* \end{array}}{\Gamma \vdash^F x = f(e_1, \dots, e_n)}$$

Function Call - Transfer

$$\frac{\begin{array}{c} \tau \text{ owner}^* f (x_1 : \tau_1, \dots, x_n : \tau_n) x_{n+1} : \tau_{n+1}, \dots, x_m : \tau_m c ; rc \in F \\ \forall 1 \leq i \leq n. \Gamma \vdash e_i : \tau_i \\ \Gamma \vdash x : \tau \text{ owner}^* \end{array}}{\Gamma \vdash^F x := f(e_1, \dots, e_n)}$$

Allocation Command

$$\frac{\Gamma \vdash x : \tau \text{ owner}^*}{\Gamma \vdash^F x := \text{new}(\tau)}$$

Deallocation Command

$$\frac{\Gamma \vdash x : \tau \text{ owner}^*}{\Gamma \vdash^F \text{delete}(x)}$$

Copy Assignment Command

$$\frac{\Gamma \vdash v : \text{int} \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash^F v = e}$$

$$\frac{\Gamma \vdash x : \tau \text{ serf}^* \quad \Gamma \vdash e : \tau \text{ serf}^*}{\Gamma \vdash^F x = e}$$

$$\frac{\Gamma \vdash x : \tau \text{ serf}^* \quad \Gamma \vdash e : \tau \text{ owner}^*}{\Gamma \vdash^F x = e}$$

Ownership Transfer Assignment Command

$$\frac{\Gamma \vdash v_1 : \tau \text{ owner}^* \quad \Gamma \vdash v_2 : \tau \text{ owner}^*}{\Gamma \vdash^F v_1 := v_2}$$

Conditional Command

$$\frac{\Gamma \vdash e : \text{int} \quad \Gamma \vdash^F c_1 \quad \Gamma \vdash^F c_2}{\Gamma \vdash^F \text{if } e \text{ then } c_1 \text{ else } c_2}$$

While Command

$$\frac{\Gamma \vdash e : \text{int} \quad \Gamma \vdash^F c}{\Gamma \vdash^F \text{while } e \text{ do } c}$$

Command Sequence

$$\frac{\Gamma \vdash^F c_1 \quad \Gamma \vdash^F c_2}{\Gamma \vdash^F c_1 ; c_2}$$

Skip Command

$$\frac{}{\Gamma \vdash^F \text{skip}}$$

Appendix C – Complete Operational Semantics for Feudal C

Definitions:

Values $\equiv \omega \cup (\text{ohr } \omega) \cup (\text{shr } \omega)$
 VarValueMap $\equiv P(\text{Loc} \rightarrow \text{Values})$

Frame $\equiv \text{VarValueMap} \times \text{Command} \times \text{Loc}$

Heap $\equiv P((\text{hr } \omega) \rightarrow \text{Values})$

States $\equiv \text{Frame list} \times \text{Heap} \times \text{Command}$

Command $\equiv \text{Com} ; \text{RetCom}$

Examples:

0 or 4 or (ohr 37) or (shr 5)

VVM = { (x, 5), (y, (shr 17)), (z, (ohr 17)) }
 or VVM = { }

$\sigma_n = \text{Frame}(\text{VVM}, c, x)$

H = { ((hr 16), 2), ((hr 30), (ohr 17)) }

S = $\langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle$

Values are natural numbers or heap references

A partial function (or set) storing the value of a set of local variables

Stack frame holds the value of all local variables, the return command sequence and return value target variable

Heap maps heap references to values

State includes the ordered list of stack frames, the heap state and the command sequence

Com and RetCom are taken directly from the syntax

Operational rule notation:

$\text{eval}(e, \sigma, H) \Downarrow v$ where $e \in \text{Aexpr}$, $\sigma \in \text{States}$, $H \in \text{Heap}$ and $v \in \text{Values}$

$\langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle \rightarrow \langle \sigma'_1, \dots, \sigma'_m \mid H' \mid c' \rangle$

where $\sigma_1, \dots, \sigma_n, \sigma'_1, \dots, \sigma'_m \in \text{Frame}$, $H, H' \in \text{Heap}$, and $c, c' \in \text{Command}$

Expression evaluation (large-step)

Command evaluation (small-step)

Shorthand:

$\sigma.\text{vvm} \equiv \text{first}(\sigma)$

where $\sigma \in \text{Frame}$

VarValueMap extractor

$\sigma.\text{cmd} \equiv \text{second}(\sigma)$

where $\sigma \in \text{Frame}$

Return command sequence extractor

$\sigma.\text{var} \equiv \text{third}(\sigma)$

where $\sigma \in \text{Frame}$

Return value target variable extractor

$\text{VVM}[v/x](y) \equiv \begin{cases} \text{VVM}(y) & \text{if } y \neq x \\ v & \text{if } y = x \end{cases}$

where $\text{VVM} \in \text{VarValueMap}$

Variable value replacement

$\sigma(x) \equiv \sigma.\text{vvm}(x)$

Local variable value lookup

$\sigma[v/x] \equiv \text{Frame}(\sigma.\text{vvm}[v/x], \sigma.\text{cmd}, \sigma.\text{var})$

Local variable value substitution

$H[v/x](y) \equiv \begin{cases} H(y) & \text{if } y \neq x \\ v & \text{if } y = x \end{cases}$

where $H \in \text{Heap}$

Heap value replacement

Helper Functions:

$\text{TC} : \text{Heap} \times (\text{hr } \omega) \rightarrow P((\text{hr } \omega))$

"Transitive Closure" - Returns all the heap addresses reachable from this heap address

$\text{TC}(H, (\text{hr } n)) \equiv \begin{cases} \{ (\text{hr } n) \} \cup \text{TC}(H, (\text{hr } x)) & \text{if } H((\text{hr } n)) = (\text{ohr } x) \\ \{ (\text{hr } n) \} & \text{otherwise} \end{cases}$

if $H((\text{hr } n)) = (\text{ohr } x)$

otherwise

$\text{del} : \text{Heap} \times (\text{ohr } \omega) \rightarrow \text{Heap}$

Deletes a chain of owner pointers in heap H, starting at address x, and returns modified heap

$\text{del}(H, (\text{ohr } \text{addr})) \equiv H - \text{TC}(H, (\text{hr } \text{addr}))$

Arithmetic Expression Evaluation

Values:

$$\frac{n \in \omega}{\text{eval}(n, \sigma, H) \Downarrow n}$$

Arithmetic Operations:

$$\frac{\text{eval}(e_1, \sigma, H) \Downarrow n_1 \quad n_1 \in \omega \quad \text{eval}(e_2, \sigma, H) \Downarrow n_2 \quad n_2 \in \omega \quad n \text{ is } n_1 \text{ op } n_2}{\text{eval}(e_1 \text{ op } e_2, \sigma, H) \Downarrow n}$$

Stack Variable Lookup:

$$\frac{x \in \text{dom}(\sigma)}{\text{eval}(x, \sigma, H) \Downarrow \sigma(x)}$$

Heap Lookup:

$$\frac{\text{eval}(v, \sigma, H) \Downarrow (\text{ohr addr}) \quad \text{val} = H(\text{hr addr})}{\text{eval}(*v, \sigma, H) \Downarrow \text{val}}$$

$$\frac{\text{eval}(v, \sigma, H) \Downarrow (\text{shr addr}) \quad \text{val} = H(\text{hr addr}) \quad \text{val} \notin (\text{ohr } \omega)}{\text{eval}(*v, \sigma, H) \Downarrow \text{val}}$$

$$\frac{\text{eval}(v, \sigma, H) \Downarrow (\text{shr addr}) \quad (\text{ohr } x) = H(\text{hr addr})}{\text{eval}(*v, \sigma, H) \Downarrow (\text{shr } x)}$$

Function Call Command (Copy)

$$\begin{aligned} & f(x_1, \dots, x_m) \quad x_{m+1}, \dots, x_p \quad c_{\text{body}} \in F_{\text{op}} \\ & \forall 1 \leq i \leq m. \text{eval}(e_i, \sigma_n, H) \Downarrow v_i \\ & \forall m+1 \leq i \leq p. v_i = 0 \end{aligned}$$

$$\langle \sigma_1, \dots, \sigma_n \mid H \mid x = f(e_1, \dots, e_m); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n, \text{Frame}(\{(x_1, v_1), \dots, (x_p, v_p)\}, c, x) \mid H \mid c_{\text{body}} \rangle$$

Function Call Command (Transfer)

$$\begin{aligned} & f(x_1, \dots, x_m) \quad x_{m+1}, \dots, x_p \quad c_{\text{body}} \in F_{\text{op}} \\ & \forall 1 \leq i \leq m. \text{eval}(e_i, \sigma_n, H) \Downarrow v_i \\ & \forall m+1 \leq i \leq p. v_i = 0 \\ & \sigma_n(x) = 0 \end{aligned}$$

$$\langle \sigma_1, \dots, \sigma_n \mid H \mid x := f(e_1, \dots, e_m); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n, \text{Frame}(\{(x_1, v_1), \dots, (x_p, v_p)\}, c, x) \mid H \mid c_{\text{body}} \rangle$$

Function Return Command

$$\forall y \in \text{dom}(\sigma_n). y \neq x \Rightarrow \sigma_n(y) \neq (\text{ohr } z)$$

$$\langle \sigma_1, \dots, \sigma_{n-1}, \sigma_n \mid H \mid \text{return } x \rangle \rightarrow \langle \sigma_1, \dots, \sigma_{n-1}[\sigma_n(x) / \sigma_n.\text{var}] \mid H \mid \sigma_n.\text{cmd} \rangle$$

Allocation Command

$$\frac{\sigma_n(x) = 0 \quad (\text{hr newaddr}) \notin \text{dom}(H)}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := \text{new}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n[(\text{ohr newaddr})/x] \mid H \cup \{((\text{hr newaddr}), 0)\} \mid c \rangle}$$

Deallocation Command

$$\frac{\sigma_n(x) = (\text{ohr addr}) \quad \forall y \in \text{dom}(\sigma_n) . y \neq x \wedge \sigma_n(y) = (\text{shr } z) \Rightarrow (\text{hr } z) \notin \text{TC}(H, (\text{hr addr}))}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{delete}(x); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n[0/x] \mid \text{del}(H, (\text{ohr addr})) \mid c \rangle}$$
$$\frac{\sigma_n(x) = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{delete}(x); c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle}$$

Skip Command

$$\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{skip}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle$$

Conditional Command

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} \neq 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{if } e \text{ then } c_1 \text{ else } c_2; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c_1; c \rangle}$$

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{if } e \text{ then } c_1 \text{ else } c_2; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c_2; c \rangle}$$

While Command

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} \neq 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{while } e \text{ do } c_{\text{loop}}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c_{\text{loop}}; \text{while } e \text{ do } c_{\text{loop}}; c \rangle}$$

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{val} = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid \text{while } e \text{ do } c_{\text{loop}}; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H \mid c \rangle}$$

Copy Assignment Command

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad x \in \text{dom}(\sigma_n)}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x = e; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n[\text{val}/x] \mid H \mid c \rangle}$$

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr addr})}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v = e; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H[\text{val}/(\text{hr addr})] \mid c \rangle}$$

$$\frac{\text{eval}(e, \sigma_n, H) \Downarrow \text{val} \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{shr addr})}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v = e; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H[\text{val}/(\text{hr addr})] \mid c \rangle}$$

Transfer Assignment Command

$$\frac{x \in \text{dom}(\sigma_n) \quad y \in \text{dom}(\sigma_n) \quad \sigma_n(y) = (\text{ohr objaddr}) \quad \sigma_n(x) = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := y ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [(\text{ohr objaddr}) / x] [0 / y] \mid H \mid c \rangle}$$

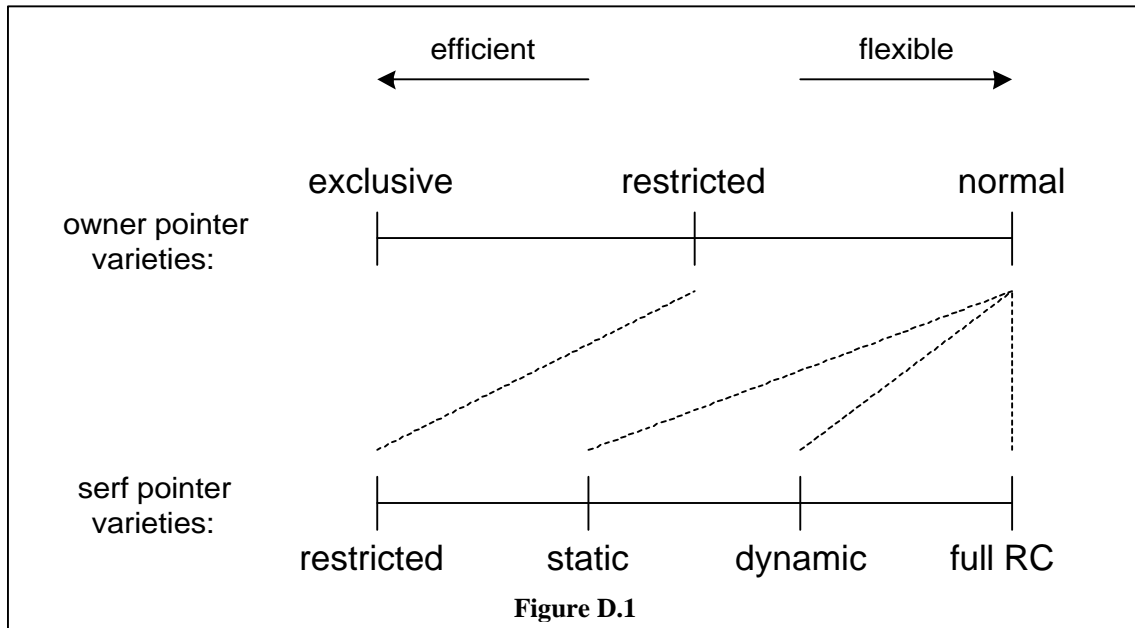
$$\frac{x \in \text{dom}(\sigma_n) \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr srcaddr}) \quad H(\text{hr srcaddr}) = (\text{ohr objaddr}) \quad \sigma_n(x) = 0}{\langle \sigma_1, \dots, \sigma_n \mid H \mid x := *v ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [(\text{ohr objaddr}) / x] \mid H [0 / (\text{hr srcaddr})] \mid c \rangle}$$

$$\frac{x \in \text{dom}(\sigma_n) \quad \text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr targetaddr}) \quad \sigma_n(x) = (\text{ohr objaddr}) \quad H((\text{hr targetaddr})) = 0 \quad (\text{hr targetaddr}) \notin \text{TC}(H, (\text{hr objaddr}))}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v := x ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n [0 / x] \mid H [(\text{ohr objaddr}) / (\text{hr targetaddr})] \mid c \rangle}$$

$$\frac{\text{eval}(v, \sigma_n, H) \Downarrow (\text{ohr targetaddr}) \quad \text{eval}(w, \sigma_n, H) \Downarrow (\text{ohr srcaddr}) \quad H(\text{hr srcaddr}) = (\text{ohr objaddr}) \quad H((\text{hr targetaddr})) = 0 \quad (\text{hr targetaddr}) \notin \text{TC}(H, (\text{hr objaddr}))}{\langle \sigma_1, \dots, \sigma_n \mid H \mid *v := *w ; c \rangle \rightarrow \langle \sigma_1, \dots, \sigma_n \mid H [(\text{ohr objaddr}) / (\text{hr targetaddr})] [0 / (\text{hr srcaddr})] \mid c \rangle}$$

Appendix D – Pointer Varieties

This appendix outlines the full range of owner and serf pointer varieties that we envision. Figure D.1 gives a snapshot idea of how the different varieties are interrelated.



Due to time limitations for the project, we chose to focus on the most restricted variety of pointers; the “restricted” owner and serf pointers. In future work, we plan to expand our inquiry to consider at least the range of types presented above. This is not meant to be a rigorous examination of these topics, merely an overview of our current ideas on new directions to explore with these different varieties.

D.1 Restricted Owner and Restricted Serf

The most basic form of owner and serf pointer is the “restricted” kind. The ideas initially presented here are greatly expanded upon in the body of the paper.

Benefits

- * No runtime cost.
- * No possibility of runtime error due to memory management error (where dereferencing null, out of memory errors, arithmetic errors, etc., are not considered memory management errors).

Restrictions

- * Only use restricted owner with restricted serf, and vice-versa.
- * Only store restricted serf on the stack (local variables).
- * Can only transfer ownership “up”.
- * Can only copy serf pointers “down” (or to another local variable in the same function).
- * Can only delete from “above”.

Assignments

- * τ restricted owner * := τ restricted owner *
- * τ restricted serf * = τ restricted owner *
- * τ restricted serf * = τ restricted serf *

D.2 Other Owner Pointer Varieties

D.2.1 Normal Owner

Benefits

- * Automatic storage reclamation, such that memory leaks cannot occur, inserted by compiler.

Restrictions

- * Exactly one owner pointer points at an object during its lifetime.

Assignments

- * τ normal owner * := new τ ; // malloc; possible runtime error: out of memory
- * τ normal owner * := τ normal owner *; // nullifies source pointer
- * τ normal owner * := NULL; // free (implicit when owner* destroyed); this may cause a runtime error if there are outstanding static, dynamic, or refct serf pointers to the object

D.2.2 Exclusive Owner

Benefits

- * No runtime overhead
- * No possibility of runtime error.
- * (design) You know only one pointer exists at a time.
- * (threading) Like a lock in some ways.

Restrictions

- * Cannot create serf pointers to it.

Assignments

- * τ exclusive owner* := new τ ;
- * τ exclusive owner* := NULL;
- * τ exclusive owner* := τ exclusive owner*;

D.3 Other Serf Pointer Varieties

D.3.1 Static Serf

Benefits

- * (performance) Only one reference bit per object.
- * Can have multiple serfs pointing at a given object.

Restrictions

- * Can only be copied “down”.
- * Only exist as a local variable.

Assignments

- * τ static serf * = τ normal owner *;
- * τ static serf * = τ static serf *; // if “down” – i.e. can be passed and copied, but not returned

Implementation

- * Creation sets the per-object reference bit, destruction restores the bit to its previous value.

D.3.2 Dynamic Serf

Benefits

- * Only one reference bit per object.
- * Can pass “up” the tree. (As opposed to static serf.)
- * Can be in a local variable, or stored in a structure, or global.

Restrictions

- * Only one dynamic serf can point at a given obj.

Assignments

- * τ dynamic serf * = τ normal owner *;
- * τ dynamic serf * = τ static serf *;
// might cause runtime error due to 2nd dyn ptr instance
- * τ dynamic serf * = τ refct serf *;
// might cause runtime error due to 2nd dyn ptr instance
- * τ dynamic serf * := τ dynamic serf *;
// nullify source (very similar to ownership.. 2nd class ownership of sorts)

D.3.3 Full Reference Count Serf

Benefits

- * No restrictions on copying (passing) or storage.

Drawbacks

- * Implementation requires a full reference count in each object that can have RC pointers to it. (There is some implementation flexibility in terms of where the RC is maintained and how large it is.)

Assignments

- * τ refct serf * = τ normal owner *;
- * τ refct serf * = τ static serf *;
- * τ refct serf * = τ dynamic serf *;
- * τ refct serf * = τ refct serf *;

Table of Contents

1 Introduction	2
1.1 Automatic Memory Management	2
1.1.1 Memory Management Errors	2
1.1.2 The State of the Art	2
1.1.3 An Attractive Alternative: The Type System	3
1.2 Design Evident in Code	3
1.3 Static Memory Management Information	3
1.4 Related Work	3
2 Fundamental Concepts	4
2.1 Owners	4
2.2 Serfs	4
2.3 The Tree	4
2.4 The Restricted Type System	4
3 Feudal C Syntax	5
3.1 General Language Information and Syntax	5
3.2 Rules of Thumb	5
3.3 Syntax	5
3.3.1 Commands	6
3.3.2 Expressions	6
3.3.3 Sample Code	6
4 Feudal C Typing Rules	7
4.1 Feudal C Typing Rules	7
4.1.1 Judgements	7
4.1.2 Commands and Expressions	7
4.1.3 Function Call and Return	8
5 Feudal C Operational Semantics	8
5.1 Notation	8
5.1.1 Semantic Entities	8
5.1.2 Program Execution	9
5.1.3 Notation	9
5.1.4 Helper Functions	9
5.2 Operational Semantics	10
5.3 Preprocessor	11
6 Proof of Memory Safety	11
6.1 Proof Introduction	11
6.1.1 Overview	11
6.1.2 General Approach	12
6.2 Theorem Statement	13
6.2.1 Theorem 1 - No Memory Leaks:	13
6.2.2 Theorem 2 - No Dangling References	13
6.3 Lemma 1 - Single Owner and No Cycles	13

6.3.1 Statement of Lemma.....	13
6.3.2 Proof of Lemma – Base Case.....	13
6.3.3 Proof of Lemma – Inductive Case.....	13
Copy Assignment Rule 1:.....	13
Copy Assignment Rule 2:.....	14
Copy Assignment Rule 3:.....	14
Transfer Assignment Rule 1:.....	14
Transfer Assignment Rule 2:.....	15
Transfer Assignment Rule 3:.....	15
Transfer Assignment Rule 4:.....	16
Allocation Rule:.....	16
Deallocation Rules:.....	16
Function Call Rule (Copy & Transfer):.....	17
Function Return Rule:.....	17
Skip, While, and Conditional Rule:.....	18
6.3.4 Proof of Lemma – Conclusion.....	18
6.4 Theorem 1 - No Memory Leaks:.....	18
6.4.1 Statement of Theorem.....	18
6.4.2 Proof of Theorem.....	18
6.5 Lemma 2 – No serfs pointing down.....	19
6.5.1 Statement of Lemma.....	19
6.5.2 Proof of Lemma – Base Case.....	19
6.5.3 Proof of Lemma – Inductive Case.....	19
Function Call (Copy) Rule:.....	19
Function Call (Transfer) Rule:.....	19
Function Return Rule:.....	19
Allocation Rule:.....	20
Deallocation Rules:.....	20
Skip, While & Conditional Rules:.....	20
Copy Assignment Rules:.....	20
Transfer Assignment Rules:.....	21
6.5.4 Conclusion.....	21
6.6 Lemma 3 – No serfs on heap.....	21
6.7 Theorem 2 - No Dangling References.....	22
6.7.1 Statement of Theorem.....	22
6.7.2 Proof of Evaluation Results Mini-Lemma.....	22
Integer values:.....	22
Arithmetic operations:.....	22
Stack Variable Lookup Rule:.....	22
Heap Lookup Rules:.....	23
6.7.3 Proof of Theorem – Base Case.....	23
6.7.4 Proof of Theorem – Inductive Case.....	23
Function Call (Copy) Rule:.....	23
Function Call (Transfer) Rule:.....	23
Function Return Rule:.....	24
Allocation Rule:.....	24
Deallocation Rules:.....	24
Skip, While & Conditional Rules:.....	25
Copy Assignment Rules:.....	25
Transfer Assignment Rules:.....	25
6.7.5 Conclusion.....	26
6.8 Theorem of Memory Safety.....	26
7 Future Work.....	27
8 Conclusion.....	27
9 Bibliography.....	27

<i>Appendix A – Complete Syntax for Feudal C</i>	28
<i>Appendix B – Complete Typing Rules for Feudal C</i>	29
<i>Appendix C – Complete Operational Semantics for Feudal C</i>	31
<i>Appendix D – Pointer Varieties</i>	35
D.1 Restricted Owner and Restricted Serf	35
D.2 Other Owner Pointer Varieties	36
D.2.1 Normal Owner.....	36
D.2.2 Exclusive Owner.....	36
D.3 Other Serf Pointer Varieties	36
D.3.1 Static Serf.....	36
D.3.2 Dynamic Serf.....	37
D.3.3 Full Reference Count Serf.....	37
<i>Table of Contents</i>	38