

# Work Related to Lighthouse

Roy Shea  
Computer Science Department,  
University of Los Angeles, California,  
roy@cs.ucla.edu

March 20, 2007

## 1 Introduction to Related Work

This paper examines the relationship between Lighthouse and current verification and analysis work within the system community.

Recent work on ownership often describes ownership using flow insensitive or flow sensitive types. At times this boundary is not perfectly clear. Flow insensitive ownership analysis tends to use simpler models that facilitate static checking of the system in question. Flow sensitive ownership analysis sit upon more robust languages with tighter ties between the language and system annotations. Lighthouse occupies a point on the design spectrum just shy of the flow sensitive analysis and uses a minimal set of program annotations. It is important to note that the novel aspects of Lighthouse do not come from new analysis techniques, but from the information that Lighthouse extracts from the systems that it examines.

A little more work remains to be done in comparing Lighthouse to other current systems research. This work will benefit from a last pass over the text to add in notes describing how Lighthouse has specifically been designed to fit the domain of reactive sensor network systems. The literature review will be rounded out with a more complete review of the inference performed by Engler style tools [9, 14]. A second look at research focusing on interprocedural analysis will become more important as finite state machines are introduced to Lighthouse.

## 2 Foundations

### 2.1 Kildall '73 With Dataflow Analysis

Kildall formalized dataflow analysis while exploring a generic framework for program optimization [13]. This early work illustrates dataflow analysis using four examples: constant propagation, common subexpression elimination, elimination of redundant register load operations, and live expression analysis. Concepts formalized in this paper, including terminology and basic theory, provide a foundation to compare current analysis using the dataflow concept. While not current research, it is valuable to cast the Lighthouse work into the system described by Kildall.

The following is description of the Lighthouse dataflow analysis. Lighthouse can be thought of as implementing two distinct dataflow analysis.<sup>1</sup> The first analysis tracks heap data within a function.

---

<sup>1</sup>In practice these two analysis can be combined. They are discussed separately to simplify the discussion.

- **Domain** Set of expressions. Expressions are in one of the states: top, empty, full with a reference to a unique allocation point, or error (bottom).<sup>2</sup>
- **Direction** Forward flow.
- **Transfer Function**
  - Allocation functions transition the receiving expression from empty to full and note the allocation location. If there is no receiving expression, a special “malloc” expression is created to track the heap data.
  - Free or store functions transition an expression from full back to empty.
  - Allocating into full expression or freeing an empty expression cause the expression to transition to error.
  - Other access to an expression causes no transition.
- **Boundary and Initialization** The dataflow initializes all program states, including the entry state, to list each expression in the top state.
- **Meet** This function’s meet operator transitions equivalent (via aliasing) expressions with different states to the error state, and leaves the state of the other expressions unchanged.
- **Dataflow Equations**

$$OUT[B] = f_b(IN[B])$$

$$IN[B] = \bigwedge_{P, pred(B)} OUT[P]$$

The second analysis uses the first analysis to track the state of persistent stores within a function.

- **Domain** Set of expressions representing stores. Stores are in one of the states: top, full, not heap, empty, or error (bottom).
- **Direction** Forwards flow.
- **Transfer Function**
  - A store operations transitions an empty store to full.
  - A free operation transitions a full store to empty.
  - Using the value of a full store does not cause a state transition.
  - Storing into a full store, freeing an empty store, or accessing the value referenced by an empty store causes the store to transition to error.
- **Boundary and Initialization** The dataflow initializes all program states to list each expression in the top state. The start state is an exception to this, and is initialized using pre-conditions (currently specified by a programmer) describing that state of the incoming stores.
- **Meet** This function’s meet operator transitions equivalent (via aliasing) expressions with different states to the error state, and leaves the state of the other expressions unchanged.
- **Dataflow Equations**

$$OUT[B] = f_b(IN[B])$$

$$IN[B] = \bigwedge_{P, pred(B)} OUT[P]$$

---

<sup>2</sup>In hindsight, a domain spanning allocation points (rather than program expressions) will probably result in a more compact analysis. The transfer functions would change appropriately.

It's amusing to note that Lighthouse's dataflows are no more complicated than the common subexpression elimination described by Kildall. Any return state in the first analysis with a full expression indicates a memory leak. An error state in either the first or second analysis indicates either an attempt to dereference an invalid pointer or inconsistent assumptions made within the flow of the program.

### 3 From Global to Interprocedural Analysis

Lighthouse is an intraprocedural analysis. Lighthouse uses annotations on function parameters to describe when a function consumes or creates heap references. While this is a safe approach, a true interprocedural analysis may provide more precise results. Works by Banning [2], Horwitz [12], and Reps [16] explore the imprecision's that occur from intraprocedural analysis and describe techniques to perform true interprocedural analysis. More in depth examination of this work remains to be done as Lighthouse formalizes its use of state machines used combine the results of separate intraprocedural analysis. This analysis is being kept brief as I focus on ownership properties that are most applicable to the current version of Lighthouse.

### 4 Typed Ownership

Inter-object aliasing in object oriented programming results in complex aliasing relationships. These related works explore techniques to controlling this inter-object aliasing to create more robust and understandable systems. Controlling these aliases relates to the exclusive ownership of heap data required in Lighthouse.

A subtle difference exists between the works described below and Lighthouse. Ownership described below statically binds ownership of an object to a dynamically created context. This naturally fits the ownership desired in object oriented systems where dynamically created objects wish to restrict ownership to their members. Lighthouse statically defines an ownership that dynamically changes at run time. This captures that changes of ownership that are important to track in the SOS operating system. Further, Lighthouse only wants to enforce uniqueness (and thus exclusive access) of heap object references across ownership boundaries. This view of uniqueness is less constrained than that observed in the system described below. Finally, the system described below were designed to handle ownership in the context of object instantiation, a concept that is not directly mappable to the modules used within SOS.

#### 4.1 Clarke '98 with Flexible Alias Protection

Clarke et. al. [4] describe ownership types for flexible alias protection within object oriented programs. This type systems adds a context type to formalize object ownership. Object contexts include a global context called `noRep` and a context local to the creating object called `rep`. On object can also describe an `owner` context separate from the objects creator. Well typed programs are guaranteed not to access an object in an inaccessible context. Typing in this system is statically verified.

Flexible alias types provide a rigorous framework of nested contexts that have access to object members. Access can escape this nesting through the `owner` context. These features make this work suitable for application to object oriented systems with member objects. The Lighthouse framework feel more linear. Heap data always has one specific owner, rather than a nested set of owners. Flexible alias protection limits alias creation to maintain type soundness. This provides very strong protection for owned data. Aliases in Lighthouse are not limited, the important invariant in Lighthouse is maintaining a single exclusive store for each heap object. Finally, a key feature provided by Lighthouse is transfer of ownership. Flexible alias protection's type system is built upon a static ownership framework.

## 4.2 Aldrich ‘02 with Alias Annotations for Program Understanding

Aldrich et. al. describe an alternate type system using alias annotations [1]. Alias annotations in their system make ownership contexts explicit and allow direct manipulation of this ownership within a program. Four alias annotations are used: owned, lent, unique, and shared. Unique objects are guaranteed to only have a single reference to them, with the exception of when they are lent to a function that. The lent annotation describes functions that may take unique objects as input, but that may have no remaining reference to the object when the function returns. Owned data encapsulates an object within the owning class. Shared data is not restricted. These annotations provide a well typed system that is implemented on top of Java. Programs are statically type checked, with some properties using being verified using standard dataflow analysis.

This work also examines program annotation inference. The inference combines rounds of constraint resolution to assign initial annotations to values, with rounds where the initial assignments flow through a functions call graph. In this manner lent / non-lent and unique annotations are inferred. A more complex analysis is used to propagate ownership constraints through potential object aliases in a function.

Lighthouse’s invariant enforced on heap data sits between the unique and owner alias annotations described in this work. A heap object within the scope of its owning module behaves like an object with the owned alias annotation. The alias annotation’s scoping of ownership is not used in Lighthouse. References to this heap object may be passed to other functions, much like the lent context, and used freely within the owning module. Transferring heap object ownership between SOS modules behaves similar to unique object references that flow between functions in this work. Both systems ensure, via a dataflow analysis, that old references to the transferred data are then dead. However, ownership transfer remains fundamentally different between Lighthouse and this system. Alias annotations allow unique pointers to flow through the program while owners of pointers can grant use of owned data to another component, but owned data may not be granted a new owner using alias annotations.

## 5 Limiting Aliases

Linear types provide a convenient means to eliminate the many referencing problems. Alias burying provides a framework for statically providing unique (or linear) variables in a program.

The primary distinction between linear types (in particular alias burying) and Lighthouse results from the strict properties enforced in the former. Unlike linear types, Lighthouse is only interested in enforced uniqueness when ownership of heap data changes. Within the context of an owning module, the Lighthouse analysis prefers to handle tracking aliases to allow more flexibility to system developers.

### 5.1 Boyland ‘01 with Alias Burying

Alias burying, described by Boyland [3], is one of multiple techniques for implementing linear references. A linear reference must either point to Null or to an unshared object. Many linear type systems accomplish this with run time destructive reads of linear references. Alias burying uses static analysis to eliminate destructive reads. This results in clearer code and the ability to statically check that aliases are treated linearly.

Alias burying allows unique objects. An intraprocedural analysis verifies that after an alias to a unique object is created, all older alias are treated as dead. This intraprocedural analysis uses a set of function annotations to capture the affect of function calls. The analysis uses function annotations to capture the effect of and verify the pre- and post- conditions each function. Shape analysis is used during this analysis to track aliases to unique objects.

Lighthouse is more permissive than alias burying. While alias burying could module exclusive ownership of heap objects passed between modules in SOS, heap objects may be freely aliased within the scope of a single SOS module. Lighthouse only tries to maintain the invariant that any heap object have exactly one exclusive owner. The reduced restrictions of Lighthouse minimize restrictions on the development of system code.

The underlying implementation of Lighthouse and alias burying appears quite similar. Dataflow analysis to ensure that all references to an object are dead after the heap object changes ownership in Lighthouse or after a new alias to a unique object is made with alias burying.

## 6 Embedded Protocols

A good deal of work has stemmed from frameworks based on the capability calculus [5] of Cray et. al. Static capabilities are used in these systems to specify permissible operations.

### 6.1 DeLine '01 with Vault

DeLine and Fähndrich describe Vault [6]. Vault associates a key with tracked resources. Operations can be guarded with a predicate of keys (that may include a key state) that must evaluate to true. The vault type system statically verifies these guards at compile time. Tracked resources are represented within the types system as a singleton type, while the type key is used in the source language to associate alias with a tracked resource. Functions changing the key set use effect clauses to describe the transitions to keys.

Vault's type system evaluates key sets for each state of a program. A simple dataflow analysis traverses each function using the effect clauses from function prototypes to update key set state based on function calls. The dataflow is initialized using the pre-conditions of the function being analyzed and is verified to match the specified post-condition at each return point in the function.

The key set analysis in Vault could be used to implement an equivalent system to the heap object tracking in Lighthouse. In such a system, each heap object would be tracked by Vault using a unique key. Ownership of heap objects would be transferred via functions that take a tracked heap object and consume (or produce) the caller's key and produce (or consume) a key in the called context. Features provided by Vault, such as multiple key states, would not be needed in this system. A problem arises from differences in ownership scoping in the two systems. Lighthouse scopes ownership of heap objects within the scope of the static store, typically corresponding to the set of functions in an SOS module. This level of scoping is not directly supported by Vault. It is not clear if this is supported in Fugue [7], a successor to Vault.

In many regards Vault goes beyond the requirements of Lighthouse. Vault increases program understanding by embedding and enforcing "best practices" information directly within source code. Lighthouse increases program understanding by providing and enforcing a simple model for data ownership. Lighthouse's simplicity may make it more amenable to program annotation inference.

### 6.2 Fähndrich '06 with the Singularity Operating System

Fähndrich et. al. created the Sing# language for fast and reliable message based communication in the Singularity operating system [10]. Singularity has a microkernel architecture made up of independent processes developed in a type-safe language with garbage collection. A separate exchange heap is used to provide fast and reliable messaging between system components. Messaging takes place over specified channels that require channel endpoints to follow a channel protocol. An exchange heap object may be owned by no more than one processes at once. Exchange heap objects are transferred between processes over the interconnecting channels. Static verification ensures that after transferring ownership of an object the process treats all references to the object as dead.

Tracking of exchange heap data in Singularity is very similar to tracking ownership of heap objects in Lighthouse. Singularity uses `TCells` to transition between static and dynamic analysis to handle non-stack references to exchange data. A `TCell` acts as a storage unit that can consume and release tracked exchange heap objects. These `TCells` share some similarities to stores used in Lighthouse. However, Lighthouse attempts to statically verify the state of stores in a program, while Singularity uses the `acquire` and `release` run time operations to manage access to a `TCell`. Singularity additionally handles tracking exchange heap data nested within exchange heap structures using the `expose` interface. Tracking of nested heap objects is not supported in Lighthouse.

Singularity provides more functionality than is sought in the the Lighthouse analysis. This functionality comes at a cost of increased code annotations and further separation of the implementation (modified C#) from standard C. Lighthouse focuses on providing guarantees to systems developed in standard C.

## 7 Analysis Tools

Members of the systems community have explored using tools, rather than language based approaches, to verify specific properties of programs. Two challenges with a tool based approach are minimizing program annotations required by the tool to reason about the system, and maximizing the applicability of the tool to multiple systems. A notable example of this work stems from the `xgcc` compiler and Metal compiler extension language used by Engler et. al. [8, 11, 9].

### 7.1 Engler '00 with Checking System Rules

Engler introduces the basic ideas of `xgcc` and Metal in [8], with more detailed descriptions of the Metal language by Hallem in [11]. This work observes that access to a compiler's internal parse tree of a program combined, with control flow information, facilitates the creation of custom verifications. Programmers write a state machine using Metal to describe a proper program property such as, "Always do X before Y". The statemachine is applied to all program paths and invalid states are reported to the user. It is important to note that the combination of `xgcc` and Metal provides a checking system, not a verification system. It is possible, for example via aliasing, for the analysis to properly identify all violations of a property in a program. Even with this limitation the tool is demonstrated to be effective at finding significant numbers of errors in real programs with a low number of false positives.

The framework described in this paper is very similar to the dataflow framework provided by CIL [15] and used in Lighthouse. While lacking the ease of the Metal language, CIL's dataflow framework facilitates the creation of very similar system checks. In contrast to the `xgcc` and Metal work, Lighthouse provides stronger guarantees to developers by focusing on eliminating false negatives. Lighthouse does this using other analysis and code permutation features included in CIL. The base `xgcc` and Metal would need to increase in complexity to facilitate these guarantees, such as allowing user directed code permutations to better track aliasing.

### 7.2 Engler '01 with Deviant Bugs

`Xgcc` and Metal easily find violations of a given specification. However, knowing specification details is a limitation of this system. For example the rule template "Always do X before Y" is easy to verify for a given X and Y. But different systems, and even different parts within a system, use different X and Y values (such as lock X protects variable Y) that may not be documented. Engler et. al. propose using must and may beliefs to overcome this limitation [9].

An example must belief is that a pointer `p` is `Null` in the `then` clause of the `if` statement `if (p==Null) then ...`. Information from must beliefs can be used to verify the internal consistency of a program. Violations of such a belief set indicate a potential bug.

A may belief assumes that that a property is true and then looks for the consistency of this belief. For example, the analysis may assume that in the statement  $f_{oo}(x); \text{tmp} = y$ ; the function  $f_{oo}$  is using the lock variable  $x$  to protect access to a later access to variable  $y$ . If this assumption is consistently followed in a program, then the assumption has a higher probability of being true. Conversely, if the assumption is not consistently followed in the program then the assumption has a higher probability of being false. Statistical analysis of different assumptions applied to a set template are performed on the code base. A violation of a may belief that is otherwise consistently followed indicates a potential failure. Kremenek et. al. [14] describe an extension to this idea that uses factor graphs to (probabilistically) reason about larger parts of a system.

They type of inference described in these works may help in generation of annotations used by the Lighthouse verification framework in a multistep framework. First Xgcc and Metal would infer interfaces conforming to a general rule, such as “Variable V can not be used after a call to function F”. Statistical analysis could find very probable violations of this rule that programmers can immediately fix. The most probably interfaces from this first step could be fed as input to a second step that uses Lighthouse to perform a more complete analysis. Again, Lighthouse aims to minimize false negatives while xgcc and Metal attempt to ease the generation of analysis.

## References

- [1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330. ACM Press, 2002.
- [2] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL*, pages 29–41. ACM Press, 1979.
- [3] J. Boyland. Alias burying: unique variables without destructive reads. *Softw. Pract. Exper.*, pages 533–553, 2001.
- [4] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64. ACM Press, 1998.
- [5] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *POPL*, pages 262–275. ACM Press, 1999.
- [6] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN*, pages 59–69, 2001.
- [7] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, 2004.
- [8] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.
- [9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP*, pages 57–72. ACM Press, 2001.
- [10] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity OS. In *EuroSys*. ACM SIGOPS, ACM, 2006.
- [11] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *The Proceedings of the 2002 PLDI*. PLDI, ACM, 2002.
- [12] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, pages 26–60, 1990.

- [13] G. A. Kildall. A unified approach to global program optimization. In *POPL*, pages 194–206. ACM Press, 1973.
- [14] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *7th USENIX OSDI*. OSDI, USENIX, 2006.
- [15] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Computational Complexity*, pages 213–228, 2002.
- [16] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61. ACM Press, 1995.