

Formal Semantics of Weak References

Kevin Donnelly J. J. Hallett Assaf Kfoury

Department of Computer Science
Boston University
{kevind,jhallett,kfoury}@cs.bu.edu

July 18, 2005

Abstract

Weak references are references that do not prevent the object they point to from being garbage collected. Many realistic languages, including Java, SML/NJ, and Haskell to name a few, support weak references. However, there is no generally accepted formal semantics for weak references. Without such a formal semantics it becomes impossible to formally prove properties of such a language and the programs written in it.

We give a formal semantics for a calculus called λ_{weak} that includes weak references and is derived from Morrisett, Felleisen, and Harper's λ_{gc} . The semantics is used to examine several issues involving weak references. We use the framework to formalize the semantics for the key/value weak references found in Haskell. Furthermore, we consider a type system for the language and show how to extend the earlier result that type inference can be used to collect reachable garbage. In addition we show how to allow collection of weakly referenced garbage without incurring the computational overhead often associated with collecting a weak reference which may be later used. Lastly, we address the non-determinism of the semantics by providing both an effectively decidable syntactic restriction and a more general semantic criterion, which guarantee a unique result of evaluation.

Categories and Subject Descriptors F3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages

General Terms Languages

Keywords Weak references, garbage collection, formal semantics

1. Introduction

1.1 Background and Motivation

Weak references are an important programming feature, supported by many modern programming languages (see the appendix of [5] for a survey of weak references in some popular languages). Weak references have shown to be particularly useful when we want to store numerous objects without allowing them to permanently occupy space. The classic examples of data structures that benefit from weak references are caches, implementations of hash-consing, and memotables [3]. In each data structure we may wish

to keep a reference to an object but also prevent that object from consuming unnecessary space. That is, we would like the object to be garbage collected once it is no longer reachable from outside the data structure despite the fact that it is reachable from within the data structure. A weak reference is the solution!

Difficulties with weak references. Despite its benefits in practice, defining formal semantics of weak references has been mostly ignored in the literature, perhaps partly because of their ambiguity and their different treatments in different programming languages. *The Weak Signature* documentation of Standard ML of New Jersey says, “The semantics of weak pointers to immutable data structures in ML is ambiguous.”[13]. The problem stems from both a lack of documentation and the intrinsic connection between weak references, garbage collection, and thus the runtime-system. The SMLofNJ Structure documentation [13] gives a slightly modified version of the following example:

```
let val (b', w') =
  let val a = (1, 2)
      val b = (1, 2)
      val w = weak(a)
    in (b, w) end
in (b', strong(w')) end
```

where `weak` and `strong` allocate and dereference weak references respectively. The types of these functions are as follows:

```
weak   : 'a → 'a weak
strong : 'a weak → 'a option.
```

After evaluation of this expression, `a` is unreachable, so one would expect the result to be `((1, 2), NONE)`. However, the object that a weak pointer references is not considered dead until garbage collection actually occurs. If the runtime-system has not initiated garbage collection then the result will be `((1, 2), SOME(1, 2))`. Also, the compiler or runtime-system may have performed subexpression elimination for optimization reasons, thus `a` and `b` would point to the same `(1, 2)`. If this is the case then `w` would remain alive as long as `b` does.

Weak references are a complex programming feature which forces the programmer to think about runtime behavior that is irrelevant without such a feature. While it would be possible to formalize a semantics for weak references without a semantics of garbage collection, such a semantics would be limited in application. Allowing the semantics of weak references to explicitly depend on garbage collection gives a more precise semantics which could, for example, let one prove more specific properties about memory usage of programs. It also forces a semantics for weak references to incorporate a semantics for garbage collection as well.

[copyright notice will appear here]

The ability to concisely specify and formally prove the correctness of garbage collection strategies, in an implementation independent way, was an important contribution of Morrisett, Felleisen and Harper’s λ_{gc} [11]. By modeling the heap as a set of mutually recursive definitions, the semantics of a garbage collection strategy can be specified as a rewrite rule which removes bindings from the mutually recursive set without altering program behavior. The addition of weak references changes this situation in that program behavior can depend on how garbage collection is employed. With such added complexity, it even more desirable to have a formal semantics.

1.2 Our Contributions and Organization of the Report

Our ultimate goal is to provide a framework for formal reasoning about weak references. We make use of the formalization to prove some properties of the language including correctness of an extended garbage collection strategy and some conditions guaranteeing the result of a program is unique regardless of when garbage collection occurs. Such a framework could also be used by implementors to prove optimal use of memory or to determine whether potential code optimizations are safe with respect to the collection of weak references.

Towards this goal, we propose a small functional programming language, λ_{weak} , using a style of definition proposed in [11]. In Section 2, we define the syntax and semantics of λ_{weak} and prove several preliminary results. A fundamental aspect of λ_{weak} is that, even though parameter-passing is deterministic (call-by-value in our case), program evaluation is non-deterministic because weak references (possibly affecting the result of the program) can be garbage-collected at any time during execution.

λ_{weak} is a particular abstract model of a functional language with weak references corresponding the weak references in Standard ML [13], but it is also flexible enough for adaptation to other languages that support weak references. In Section 3 we show how λ_{weak} can be adapted to model the weak references found in Haskell, and outline appropriate changes in order to handle the case of Java.

In Section 4 we set up a type system for λ_{weak} which, in addition to enforcing the standard invariants, i.e., catching programs that “go wrong” (Subsection 4.1), can be used for a more efficient management of memory (Subsection 4.2). We extend the previous result by showing that we can use type inference to allow for the collection of additional weak references without incurring the runtime penalty that might otherwise occur if a collected weak reference is later used.

In Section 5 we study the conditions under which λ_{weak} programs are “well-behaved”, i.e., under which garbage collection does not affect the result of evaluation. It is undecidable whether an arbitrary λ_{weak} program is well-behaved.

In Subsection 5.1 we define a proper subclass of well-behaved programs that is efficiently recognizable (in linear time) and encompasses some common uses of weak references. In Subsection 5.2 we define a sufficient (but not necessary) general criterion for the well-behavedness of programs which can be used as a guideline for writing programs satisfying the property.

In Section 6 we discuss related work and in Section 7, we propose several directions for future research and conclude. Missing proofs and additional materials can be found in the two companion reports, [4] and [5].

2. Modeling Weak References: λ_{weak}

A formal model, which extends λ_{gc} with the means to introduce and conditionally dereference weak references, is introduced in [5] and further investigated in [4]. The semantics given essentially matches the semantics for weak references in SML/NJ [13]. Be-

cause of compiler optimizations like common subexpression elimination, the actual identity of immutable objects is not statically known, and as such the documentation claims the semantics is “ambiguous.” However, if an SML programmer uses our semantics in a program, it will behave as expected (regardless of common subexpression elimination) since our semantics does not guarantee that any reference will be collected at any particular time. This is a natural restriction as the programmer (usually) has no control over if or when garbage collection occurs.

Syntax of λ_{weak}

The syntax of λ_{weak} (given in Figure 1) is that of a standard programming language based on the λ -calculus along with additional primitives for introducing weak references and doing conditional weak dereferencing. A λ_{weak} expression is either a variable (x), an integer (i), a pair ($\langle e_1, e_2 \rangle$), a projection ($\pi_i e$), an abstraction ($\lambda x. e$), an application ($e_1 e_2$), a weak expression ($weak e$) or an ifdead expression ($ifdead e_1 e_2 e_3$).

Heap values, hv , are values which may be allocated to the heap during reduction. Heap values are a subset of expressions in addition to the special value d (meaning “dead”). During execution, a weak pointer “ $weak y$ ” on the heap may be replaced with d if the only remaining references to y are weak.

A λ_{weak} program, $letrec H in e$ consists of a set of mutually recursive definitions (given by a finite map $H : Var \rightarrow Hval$) which models the heap, and an expression e . We write $H \uplus H'$ to be the union of two heap functions defined on disjoint domains and $Dom(H)$ to be the domain of H and we define

$$H^s = \{x \mapsto H(x) \mid H(x) \neq weak y \text{ for any } y\}$$

to be the strong part of the heap. The set of free variables of an expression, $FV(e)$ and capture-avoiding substitution $e\{x := e'\}$ are defined as usual. Free variables for a heap H and a program $letrec H in e$ are defined by:

$$FV(H) = \left(\bigcup_{x \in Dom(H)} FV(H(x)) \right) - Dom(H)$$

$$FV(letrec H in e) = (FV(H) \cup FV(e)) - Dom(H)$$

Expressions are identified up to α -conversion and programs are identified up to renaming of variables bound in the heap, e.g., $letrec H \uplus \{x \mapsto h\}$ in $x = letrec H \uplus \{y \mapsto h\}$ in y assuming $x \notin FV(H)$ and $y \notin FV(H)$.

Semantics of λ_{weak}

The reduction semantics of λ_{weak} are given by the evaluation contexts (which apply left-to-right, call-by-value reduction) and rewrite rules in Figure 1. We use the following notation for rewrite rules. Let G be a set of rules and P and P' be programs:

$P \xrightarrow{G} P'$ means P rewrites to P' by some rule in G and $\xrightarrow{G^*}$ is the reflexive, transitive closure of \xrightarrow{G} .

$P \Downarrow_G P'$ means $P \xrightarrow{G^*} P'$ and P' is irreducible with respect to the rules in G .

$P \Uparrow_G$ means there exists an infinite reduction using rules in G starting from program P .

The evaluation rules are chosen to extend normal evaluation with reference values and weak references. The rule (alloc) allocates a value on the heap and replaces it with a reference. The rule (app) evaluates function calls by reference passing. In this language, all values are “reference values” in that they are allocated to the heap and passed by reference. The projection rules (π_i) extract the appropriate component from a pair pointed to by a reference. Rule (ifdead) applied to $P = letrec H in E[ifdead x e_2 e_3]$

Programs:			
(variables)	$w, x, y, z \in \text{Var}$		
(integers)	$i \in \text{Int}$	$::= \dots -2 -1 0 1 2 \dots$	
(expressions)	$e \in \text{Exp}$	$::= x i \langle e_1, e_2 \rangle \pi_1 e \pi_2 e \lambda x. e e_1 e_2 $	$\text{weak } e \text{ifdead } e_1 e_2 e_3$
(heap values)	$hv \in \text{Hval}$	$::= i \langle x_1, x_2 \rangle \lambda x. e \text{weak } x \mathbf{d}$	
(heaps)	$H \in \text{Var} \xrightarrow{\text{fin}} \text{Hval}$		
(programs)	$P \in \text{Prog}$	$::= \text{letrec } H \text{ in } e$	
(answers)	$A \in \text{Ans}$	$::= \text{letrec } H \text{ in } x$	

Evaluation Contexts and Instruction Expressions:			
(contexts)	$E \in \text{Ctxt}$	$::= [] \langle E, e \rangle \langle x, E \rangle \pi_i E E e x E \text{weak } E \text{ifdead } E e_1 e_2$	
(instruction)	$I \in \text{Instr}$	$::= hv \pi_i x x y \text{ifdead } x e_1 e_2$	

Rewrite Rules:			
(alloc)	$\text{letrec } H \text{ in } E[hv] \xrightarrow{\text{alloc}} \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } E[x]$	where x is a fresh variable	
(π_i)	$\text{letrec } H \text{ in } E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H \text{ in } E[x_i]$	provided $H(x) = \langle x_1, x_2 \rangle$ and $i \in \{1, 2\}$	
(app)	$\text{letrec } H \text{ in } E[x y] \xrightarrow{\text{app}} \text{letrec } H \text{ in } E[e\{z := y\}]$	provided $H(x) = \lambda z. e$	
(ifdead)	$\text{letrec } H \text{ in } E[\text{ifdead } x e_1 e_2] \xrightarrow{\text{ifdead}}$	$\begin{cases} \text{letrec } H \text{ in } E[e_2 w] & \text{if } H(x) = \text{weak } w \\ \text{letrec } H \text{ in } E[e_1] & \text{if } H(x) = \mathbf{d} \end{cases}$	

Figure 1. Syntax and Operational Semantics of λ_{weak}

does a conditional deallocation of weak reference x . If $H(x) = \text{weak } y$ (the weak reference is not dead) then P reduces to $\text{letrec } H \text{ in } E[e_3 y]$. If $H(x) = \mathbf{d}$ then P reduces to $\text{letrec } H \text{ in } e_2$.

There is an additional rewrite rule (`garb`) not listed in Figure 1 which uses the following as auxiliary rules.

- | | | | |
|-----------|--|--|--|
| (gc) | $\text{letrec } H_1 \uplus H_2 \text{ in } e \xrightarrow{\text{gc}} \text{letrec } H_1 \text{ in } e$ | provided $\text{Dom}(H_2) \cap FV(\text{letrec } H_1^s \text{ in } e) = \emptyset$,
and $H_2 \neq \emptyset$ | |
| (weak-gc) | $\text{letrec } H \uplus \{x \mapsto \text{weak } y\} \text{ in } e \xrightarrow{\text{weak-gc}}$ | $\text{letrec } H \uplus \{x \mapsto \mathbf{d}\} \text{ in } e$
provided $y \notin \text{Dom}(H)$ | |

Using these rules we define the garbage collection rule (`garb`) as follows:

- | | | | |
|--------|--|--|--|
| (garb) | $\text{letrec } H \text{ in } e \xrightarrow{\text{garb}} \text{letrec } H' \text{ in } e$ | provided $\text{letrec } H \text{ in } e \xrightarrow{\text{gc}} \text{letrec } H'' \text{ in } e$
and $\text{letrec } H'' \text{ in } e \Downarrow_{\text{weak-gc}} \text{letrec } H' \text{ in } e$ | |
|--------|--|--|--|

Intuitively the rule (`garb`) works by first collecting some bindings to which there is no strong reference, then setting to dead all the weak references which refer to collected bindings. Notice that this rewrite rule allows for the collection of cycles of garbage¹. Often, in practice garbage collection will collect every location to which there is no strong reference, however we do not want the programmer to rely on this behavior, so the rule reflects this. In particular, it often makes sense for the garbage collector to try to not collect weakly reachable references if there is not a shortage of memory. By using this rule we allow the implementor of the garbage collector complete freedom as to what garbage is collected as long as weak references to collected locations are all properly tombstoned (which is reflected by the $\Downarrow_{\text{weak-gc}}$ in the rule). In addition, using this

¹An easier-to-understand but more restrictive definition of (`gc`) is

$$\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e \xrightarrow{\text{gc}} \text{letrec } H \text{ in } e$$

provided $x \notin FV(\text{letrec } H^s \text{ in } e)$

This rule collects only one binding at a time and does not permit the collection of cycles of garbage. Having thus redefined (`gc`), “ $\xrightarrow{\text{gc}}$,” should also be replaced by “ $\xrightarrow{\text{gc}}^*$ ” in the definition of (`garb`).

rule we are sure that the programmer cannot rely on some location having been collected, so it is safe to perform compiler optimizations which make object identity statically unknown (like common subexpression elimination).

We denote the set of rewrite rules by

$$R = \{\text{alloc}, \pi_1, \pi_2, \text{app}, \text{ifdead}, \text{garb}\}.$$

Given the rewrite rule (`garb`), the reduction is no longer confluent because the initiation of garbage collection can effect the reduction of `ifdead` expressions. The example program P_1 shown in Figure 2, taken from [5], shows the non-confluence of λ_{weak} .

We can even have a program whose behavior can converge or diverge depending on when garbage collection occurs. For example:

$$P_2 = \text{letrec } \{\} \text{ in } (\lambda x. \text{ifdead } (\text{weak } x) 0$$

$$(\text{ifdead } (\text{weak } x) \Omega (\lambda z. \lambda y. \pi_i y)))$$

where $\Omega = (\lambda y. y y)(\lambda y. y y)$.

If a program is completely evaluated without getting stuck, it will have to form $\text{letrec } H \text{ in } x$. In order to be able to formally talk about the result of the evaluation, we define $\text{result}(H, e)$ as in Figure 3. We use this to talk about the final result since there may be syntactically different values which have the same value.

An irreducible value is either an answer, $\text{letrec } H \text{ in } x$, or a stuck program which corresponds to an error.

Definition 2.2 (Stuck Programs). A λ_{weak} program is stuck if it is of one of the following forms:

$$\text{letrec } H \text{ in } E[\pi_i x]$$

$$(x \notin \text{Dom}(H) \text{ or } H(x) \neq \langle x_1, x_2 \rangle)$$

$$\text{letrec } H \text{ in } E[x y]$$

$$(x \notin \text{Dom}(H) \text{ or } H(x) \neq \lambda z. e)$$

$$\text{letrec } H \text{ in } E[\text{ifdead } x e_1 e_2]$$

$$(x \notin \text{Dom}(H) \text{ or } (H(x) \neq \text{weak } w \text{ and } H(x) \neq \mathbf{d})) \quad \square$$

EXAMPLE 2.1.

$P_1 = \text{letrec } \{ \} \text{ in } (\lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y)) \langle 5, 6 \rangle$
 $\xrightarrow{\text{alloc}}$ $\text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y) \} \text{ in } a \langle 5, 6 \rangle$
 $\xrightarrow{\text{alloc}}$ $\text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5 \} \text{ in } a \langle b, 6 \rangle$
 $\xrightarrow{\text{alloc}}$ $\text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6 \} \text{ in } a \langle b, c \rangle$
 $\xrightarrow{\text{alloc}}$ $\text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle \} \text{ in } a e$
 $\xrightarrow{\text{app}}$ $\text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle \} \text{ in ifdead } (\text{weak } e) 0 (\lambda y. \pi_1 y)$
 $\xrightarrow{\text{alloc}}$ $\text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle, f \mapsto \text{weak } e \} \text{ in ifdead } f 0 (\lambda y. \pi_1 y)$
 $\xrightarrow{\text{then}}$ $\text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle, f \mapsto \text{weak } e \} \text{ in } (\lambda y. \pi_1 y) e \longrightarrow \dots \xrightarrow{\text{garb}}$ $\text{letrec } \{ b \mapsto 5 \} \text{ in } b$
 or
 $\xrightarrow{\text{garb}}$ $\text{letrec } \{ f \mapsto d \} \text{ in ifdead } f 0 (\lambda y. \pi_1 y) \longrightarrow \dots \xrightarrow{\text{garb}}$ $\text{letrec } \{ g \mapsto 0 \} \text{ in } g$
 where a, b, c, e, f and g are fresh variables introduced in the process of program evaluation. \square

Figure 2. Example of Non-confluent Reduction

$$\text{result}(H, e) = \begin{cases} x & \text{if } e = x \text{ and } x \notin \text{Dom}(H) \\ \text{result}(H, H(x)) & \text{if } e = x \text{ and } x \in \text{Dom}(H) \\ i & \text{if } e = i \\ d & \text{if } e = d \\ \langle \text{result}(H, e_1), \text{result}(H, e_2) \rangle & \text{if } e = \langle e_1, e_2 \rangle \\ \pi_i \text{ result}(H, e') & \text{if } e = \pi_i e' \text{ and } i \in \{1, 2\} \\ \lambda x. \text{result}(H, e') & \text{if } e = \lambda x. e' \text{ where } x \notin \text{Dom}(H) \\ \text{result}(H, e_1) \text{ result}(H, e_2) & \text{if } e = e_1 e_2 \\ \text{weak result}(H, e') & \text{if } e = \text{weak } e' \\ \text{ifdead result}(H, e_1) \text{ result}(H, e_2) \text{ result}(H, e_3) & \text{if } e = \text{ifdead } e_1 e_2 e_3 \end{cases}$$

Figure 3. Definition of result(H, e)

Definition 2.3 (Evaluation Set). The evaluation set of a program P relative to a set of rewrite rules G :

$$\text{eval-set}(P, G) = \{ \perp \mid P \uparrow_G \} \cup \{ \text{error} \mid P \downarrow_G P' \text{ and } P' \text{ is stuck} \} \cup \{ \text{result}(H, x) \mid P \downarrow_G \text{letrec } H \text{ in } x \}$$

\square

In general, $\text{eval-set}(P, G)$ is an undecidable set. If $G = R$, we write $\text{eval-set}(P)$ instead of $\text{eval-set}(P, R)$. For the programs in the previous examples, we have $\text{eval-set}(P_1) = \{0, 5\}$ and $\text{eval-set}(P_2) = \{0, 5, \perp\}$.

Definition 2.4 (Program Equivalence). $(P, G) \equiv (P', G')$ iff $\text{eval-set}(P, G) = \text{eval-set}(P', G')$. If $G = G' = R$, we simply write $P \equiv P'$. \square

Note that our program equivalence “ \equiv ” is more general than Kleene equivalence “ \simeq ” used in [11]. However, if P and P' are well-typed with return type `int` and do not use weak references, then $P \equiv P'$ iff $P \simeq P'$. Kleene equivalence is not sufficient to formally describe “equivalent behavior” of λ_{weak} programs which can have more than one possible outcome.

We define garbage using program equivalence. Any binding which does not contribute to the final result is garbage.

Definition 2.5 (Garbage). Let $P = \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e$. Then the binding “ $x \mapsto hv$ ” is garbage in P iff $P \equiv \text{letrec } H \text{ in } e$. \square

Proposition 2.6. *It is undecidable whether a binding is garbage in an arbitrary program.*

Proof sketch. Consider the program

$$P = \text{letrec } \{x \mapsto 5\} \text{ in } (\lambda y. x) e,$$

where e is an arbitrary closed lambda-expression. The binding “ $x \mapsto 5$ ” is garbage in P iff e diverges according to call-by-value β -reduction, which is undecidable. \square

Definition 2.7 (Well-Behaved Programs). A program, P , is well-behaved iff $\text{eval-set}(P)$ is a singleton set – in words, iff either all evaluations of P diverge, or all evaluations of P get stuck, or all evaluations of P converge and return the same result. \square

Proposition 2.8. *It is undecidable whether an arbitrary program is well-behaved.*

Proof sketch. Let e be an arbitrary closed lambda-expression, $\Omega = \omega \omega$ and $\omega = (\lambda x. x x)$. Then

$$\text{letrec } \{ \} \text{ in ifdead } (\text{weak } 5) \Omega (\lambda x. e)$$

is well-behaved if and only if e diverges according to call-by-value β -reduction. \square

The two preceding propositions, though not difficult to prove, frame the rest of the discussion in the paper.

Proposition 2.6 shows it is impossible to compute an optimal garbage collection strategy, i.e., one that removes all garbage from the heap. Thus, any gc algorithm must conservatively approximate bindings that are garbage.

Proposition 2.8 shows it is impossible to recognize exactly the set of programs evaluating to unique results, so that any (decidable) criterion for this property must conservatively approximate well-behaved programs.

3. Modeling Other Forms of Weak References

Our formal setup is flexible enough to handle weak references in other popular programming languages, including Haskell and Java.

3.1 Haskell Key/Value Weak References

In this section we formalize the key/value weak references found in Haskell (which are similar to ephemerons [6]). We call the garbage collection rule derived from the Haskell documentation (garb'). As an application in the absence of side-effects, we define a more "aggressive" gc rule, called (garb''), and prove its correctness in Theorem 3.1.

A key/value weak reference is a special type of weak reference which contains both a key and a value. To simplify things we do not consider finalizers (which are included in Haskell). During garbage collection, the tracer does not trace the value of a weak pointer unless the key is otherwise reachable. Such references are a generalization of ordinary weak references which are used in creating weak mappings with complex collection behavior, like memotables (as described in [8]). In the GHC documentation [7], the semantics is specified as follows.

Informally, something is reachable if it can be reached by following ordinary pointers from the root set, but not following weak pointers. We define reachability more precisely as follows A heap object is reachable if:

- It is directly pointed to by a reachable object, other than a weak pointer object.
- It is a weak pointer object whose key is reachable.
- It is the value or finaliser of an object whose key is reachable.

Notice that a pointer to the key from its associated value or finaliser does not make the key reachable. However, if the key is reachable some other way, then the value and the finaliser are reachable, and so, therefore, are any other keys they refer to directly or indirectly.

We replace the syntax $\text{weak } e$ with $\text{KVweak}(e_1, e_2)$ where e_1 is the key and e_2 is the value. In order to specify the reachable parts of the heap we define the one step closure of H with respect to H' (where $H \subseteq H'$) by:

$$\begin{aligned} C_{H'}(H) &= H \cup \{z \mapsto H'(z), x \mapsto \text{KVweak}(y, z) \mid \\ &\quad \exists x \in \text{Dom}(H'). \exists y \in \text{Dom}(H). \\ &\quad H'(x) = \text{KVweak}(y, z)\} \end{aligned}$$

We define the reachable part of the heap,

$$R(H, e) = \bigcup_{n \in \mathbb{N}} C_H^{(n)}(H \upharpoonright FV(e))$$

where $f \upharpoonright S$ means the restriction of f to domain $S \cap \text{Dom}(f)$ and $f^{(n)}$ means composition of n copies of f . It is easy to see that this definition of reachability meets the definition given in the Haskell documentation. We get rid of the reduction rule (garb) and use the following instead.

$$\begin{aligned} (\text{gc}') \quad &\text{letrec } H_1 \uplus H_2 \text{ in } e \xrightarrow{\text{gc}'} \text{letrec } H_1 \text{ in } e \\ &\text{provided } \text{Dom}(H_2) \cap \text{Dom}(R(H_1 \uplus H_2, e)) = \emptyset \\ &\text{and } H_2 \neq \emptyset \end{aligned}$$

We still make use of (essentially) the original (weak-gc) rule

$$\begin{aligned} (\text{weak-gc}) \quad &\text{letrec } H \uplus \{x \mapsto \text{KVweak}(y, z)\} \text{ in } e \xrightarrow{\text{weak-gc}} \\ &\text{letrec } H \uplus \{x \mapsto d\} \text{ in } e \\ &\text{provided } y \notin \text{Dom}(H) \end{aligned}$$

We then define our new garbage collection rule (garb') by

$$\begin{aligned} (\text{garb}') \quad &\text{letrec } H \text{ in } e \xrightarrow{\text{garb}'} \text{letrec } H' \text{ in } e \\ &\text{provided } \text{letrec } H \text{ in } e \xrightarrow{\text{gc}'} \text{letrec } H'' \text{ in } e \\ &\text{and } \text{letrec } H'' \text{ in } e \Downarrow_{\text{weak-gc}} \text{letrec } H' \text{ in } e \end{aligned}$$

The semantics given in Haskell causes a key/value weak pointer to be reachable if its key is reachable, even if the weak pointer object itself is unreachable. The reason for this is to maintain the guarantee that finalizers are run exactly once. Because we do not have side-effects and finalizers in our language we can simplify the semantics by using the following definition of the one step closure of H with respect to H' .

$$\begin{aligned} C_{H'}(H) &= H \cup \{z \mapsto H'(z) \mid \exists x, y \in \text{Dom}(H). \\ &\quad H(x) = \text{KVweak}(y, z)\} \end{aligned}$$

We then define the reachable heap by

$$R'(H, e) = \bigcup_{n \in \mathbb{N}} C_H^{(n)}(H \upharpoonright FV(e))$$

which requires that both a weak pointer object and its key be reachable for the value to be reachable. This definition allows for the collection of more garbage. We can define garbage collection rules which use this definition of reachability.

$$\begin{aligned} (\text{gc}'') \quad &\text{letrec } H_1 \uplus H_2 \text{ in } e \xrightarrow{\text{gc}''} \text{letrec } H_1 \text{ in } e \\ &\text{provided } \text{Dom}(H_2) \cap \text{Dom}(R'(H_1 \uplus H_2, e)) = \emptyset \\ &\text{and } H_2 \neq \emptyset \\ (\text{garb}'') \quad &\text{letrec } H \text{ in } e \xrightarrow{\text{garb}''} \text{letrec } H' \text{ in } e \\ &\text{provided } \text{letrec } H \text{ in } e \Downarrow_{\text{gc}''} \text{letrec } H'' \text{ in } e \\ &\text{and } \text{letrec } H'' \text{ in } e \Downarrow_{\text{weak-gc}} \text{letrec } H' \text{ in } e \end{aligned}$$

Let

$$R' = \{\text{alloc}, \pi_1, \pi_2, \text{app}, \text{ifdead}, \text{garb}'\}$$

and

$$R'' = \{\text{alloc}, \pi_1, \pi_2, \text{app}, \text{ifdead}, \text{garb}''\}.$$

Then we have the following.

Theorem 3.1. *Let e be an arbitrary expression in the key/value weak calculus. Then*

$$\begin{aligned} &\text{letrec } H \text{ in } e \xrightarrow{R'}^* \text{letrec } H' \text{ in } x \\ \text{iff } &\text{letrec } H \text{ in } e \xrightarrow{R''}^* \text{letrec } H'' \text{ in } x \end{aligned}$$

with $\text{result}(H', x) = \text{result}(H'', x)$.

Proof sketch. The only difference between (garb') and (garb'') is that (garb'') may garbage collect more bindings that are not reachable according to the standard free-variable definition of reachability (i.e. x reachable iff $x \in FV(\text{letrec } H \text{ in } e)$). Since the transition rules (excluding gc rules) only branch on reachable variables and the definition of result only depends on reachable variables, these differences do not effect the result of the program. \square

3.2 Java Soft and Weak References

Java provides several types of weak references with different interactions with the garbage collector [14]. Phantom references (`java.lang.ref.PhantomReference`) cannot be dereferenced and are merely a finalization mechanism, so we do not consider them here. Soft references (`java.lang.ref.SoftReference`) and Weak references (`java.lang.ref.WeakReference`) are two forms of weak references with different garbage collection behavior.

Soft references are weak references which the garbage collector tries to keep alive, even after the object is no longer strongly reachable, as long as there is enough memory to do so. Soft references

are intended to be used in implementing data caches in, for example, memoized functions. The system makes no guarantee about when soft references will be collected.

Weak references are weaker than soft references in that the garbage collector will reclaim objects weakly referenced as soon as there are no more strong references, even if memory is not low. Weak references in Java are intended to be used for canonicalizing mappings such as hash-consing lists.

It is fairly straightforward to extend our calculus to formalize Java's weak references. We just need two types of weak references which are treated differently by the garbage collection rule.

4. Types for Garbage Collection

As in [11] we introduce a standard monomorphic type system, and show how one can use type inference to collect additional garbage. Additionally, we extend this result to allow for collecting additional weakly-referenced garbage without incurring the overhead of re-computing data cached in the weak references.

4.1 Monomorphic Type System

In this section we introduce a standard monomorphic type system for λ_{weak} . There are no surprises here. While we could formulate an explicitly typed language allowing for tag-free garbage collection [1, 15], this was already done in [11] and there are no additional complications arising from weak references. Therefore we will formulate an implicit type assignment system for the language λ_{weak} already defined.

The syntax of types is as follows.

Types:

(types) $\tau \in \text{Type} ::= \text{int} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \tau \text{ weak}$

The typing rules are shown in Figure 4 and are (almost) completely standard. The one non-standard addition is that we assign to the type $\tau \text{ weak}$ for any τ . This is necessary for type assignments to be preserved by the rule (weak-gc).

This type system is sound, it rules out stuck programs, which is proven using the following progress and preservation lemmas [16], whose proofs are standard.

Lemma 4.1 (Progress). *For every λ_{weak} program P , if $\vdash P : \tau$ then either P is an answer or there exists P' such that $P \xrightarrow{R} P'$.* \square

Lemma 4.2 (Preservation). *For every λ_{weak} program P , if $\vdash P : \tau$ and $P \xrightarrow{R} P'$ then $\vdash P' : \tau$.* \square

Theorem 4.3 (Type Soundness). *For every λ_{weak} program P , if $\vdash P : \tau$ then either P is an answer or else there is some P' such that $P \xrightarrow{R} P'$ and $\vdash P' : \tau$.* \square

Further discussion of the type system along with an efficient type inference algorithm can be found in [5].

4.2 Collecting Reachable Garbage

As was pointed out in [2], and proven in [11], one can use type inference to detect that the values of certain references will never be used. Any binding that will never be used is semantically garbage regardless of whether or not it is reachable. So reachable values that will never be used can be changed to any value (we use 0) without affecting the result of the program. This can allow for additional garbage collection. For example the program

$\text{letrec } \{x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto \langle x_2, x_2 \rangle, x_4 \mapsto \langle x_1, x_3 \rangle\}$ in $\pi_1 x_4$

is equivalent to the program

$\text{letrec } \{x_1 \mapsto 1, x_3 \mapsto 0, x_4 \mapsto \langle x_1, x_3 \rangle\}$ in $\pi_1 x_4$

so we can safely collect the binding $x_2 \mapsto 2$.

This is formalized by considering the base language, in our case λ_{weak} , to be an implicitly typed language with type variables.

(types) $\tau \in \text{Type} ::= t \mid \text{int} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \tau \text{ weak}$

The presence of type variables allows us to derive a *most general typing* for each well-typed program, i.e., a typing such that every other typing of the same program is a substitution instance of it.

We prove a slightly stronger version of preservation for this system. We will use this preservation theorem to prove that if a binding can be assigned a type variable then after a reduction step that binding can still be assigned a type variable.

Lemma 4.4 (Preservation). *If there exists a typing $\Gamma \vdash e : \tau$ and for some $\vdash H : \Gamma$, we have $\text{letrec } H$ in $e \xrightarrow{R} \text{letrec } H'$ in e' then there exists Γ' with $\vdash H' : \Gamma'$, and $\Gamma' \vdash e' : \tau$ such that for all $x \in (\text{Dom}(\Gamma) \cap \text{Dom}(\Gamma'))$ we have $\Gamma(x) = \Gamma'(x)$.*

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$. \square

We then use type inference to generate a most general typing for a given program. If we are ever able to assign a type variable to a reference, then the value of this reference cannot affect the result of the program. In order to prove this we will first define the active positions of a term (which are the occurrences that constrain the type of a reference).

Definition 4.5. We say x occurs in an *active position* of e if one of the following occurs as a subterm of e :

1. $x e'$ for some e' , or
2. $\pi_i x$, or
3. $\text{ifdead } x e_1 e_2$ for some e_1, e_2 . \square

In any typing derivation, no reference that is assigned a type variable may appear in an active position.

Lemma 4.6. *If $\Gamma \uplus \{x : t\} \vdash e : \tau$ then x does not occur in an active position in e .*

Proof. It is easy to see that each typing rule whose conclusion creates a new active position ((proj_i), (app), and (ifdead)) constrains the type of the term appearing in the active position to be something other than a type variable. \square

The addition of the type $\tau \text{ weak}$ only slightly affects this basic result. Because garbage collection can affect the result of a program we assume that we are working with a well-behaved program. There is no problem with doing the extra inference-based collection on non-well-behaved programs, but the collected program is not equivalent to the original since its behaviors may be a proper subset of the behaviors of the original program. The following proof technique is due to Morrisett²[10].

Theorem 4.7 (Inference GC). *Let*

$$\begin{aligned} \Gamma_1 &= \{x_1 : t_1, \dots, x_n : t_n\}, \\ H_1 &= \{x_1 \mapsto h_1, \dots, x_n \mapsto h_n\}, \text{ and} \\ H'_1 &= \{x_1 \mapsto 0, \dots, x_n \mapsto 0\}. \end{aligned}$$

If

1. $\Gamma_1 \uplus \Gamma_2 \vdash e : \tau$ ($\tau \notin Tvar$), and
2. $\Gamma_1 \vdash H_2 : \Gamma_2$, and
3. $\exists S.\emptyset \vdash H_1 : S\Gamma_1$, and

²As pointed out by Morrisett, the same proof technique can be used to establish a similar result, Theorem 5.3 in [11], whose original proof was a far more complicated argument using logical relations.

$$\begin{array}{c}
\overline{\Gamma \uplus \{x : \tau\} \vdash x : \tau} \text{ (var)} \quad \overline{\Gamma \vdash i : \text{int}} \text{ (int)} \quad \overline{\Gamma \vdash d : \tau \text{ weak}} \text{ (dead)} \\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \text{ (pair)} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i e : \tau_i} \text{ (proj}_i\text{)} \text{ (for } i = 1, 2\text{)} \\
\frac{\Gamma \uplus \{x : \tau_1\} \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \text{ (abs)} \quad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \text{ (app)} \\
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{weak } e : \tau \text{ weak}} \text{ (weak)} \quad \frac{\Gamma \vdash e_1 : \tau_1 \text{ weak} \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e_3 : \tau_1 \rightarrow \tau_2}{\Gamma \vdash \text{ifdead } e_1 e_2 e_3 : \tau_2} \text{ (ifdead)} \\
\frac{\forall x \in \text{Dom}(\Gamma'). \Gamma \uplus \Gamma' \vdash H(x) : \Gamma'(x)}{\Gamma \vdash H : \Gamma'} \text{ (heap)} \quad \frac{\emptyset \vdash H : \Gamma \quad \Gamma \vdash e : \tau}{\vdash \text{letrec } H \text{ in } e : \tau} \text{ (prog)}
\end{array}$$

Figure 4. Typing rules for λ_{weak}

4. $\text{letrec } H_1 \uplus H_2 \text{ in } e$ is well behaved (i.e. the timing of garbage collection cannot affect the final result)

then $\text{letrec } H_1 \uplus H_2 \text{ in } e \equiv \text{letrec } H'_1 \uplus H_2 \text{ in } e$.

Proof sketch. Since we are dealing with a well-behaved program, we can ignore the (garb) rule for the purpose of showing equivalence. Since, the other rules only add bindings, we have that if

$$\text{letrec } H_1 \uplus H_2 \text{ in } e \xrightarrow{\text{R-}\{\text{garb}\}}^* \text{letrec } H_1 \uplus H_2 \uplus H_3 \text{ in } e'$$

then for any $\Gamma_1 \uplus \Gamma_2 \vdash H_3 : \Gamma_3$ we have $\Gamma_1 \uplus \Gamma_2 \uplus \Gamma_3 \vdash e' : \tau$ by Theorem 4.4. By Lemma 4.6, none of x_1, \dots, x_n can appear in an active position in e' . The reduction rules only depend on the value of references in an active position, so we will never reduce to a state whose next transition depends on the value of any of x_1, \dots, x_n , therefore $\text{letrec } H_1 \uplus H_2 \text{ in } e \equiv \text{letrec } H'_1 \uplus H_2 \text{ in } e$. \square

So type-inference based GC works in this language, however we have introduced a new potential problem. The problem is that often weak pointers are often used to cache data that was computationally expensive to produce, so killing a weak pointer may cause unnecessary recomputation. Consider the following program:

$$\text{letrec } \{x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto \langle x_2, x_2 \rangle, x_4 \mapsto \langle x_1, x_3 \rangle, \\ x_5 \mapsto \text{weak } x_2, f \mapsto \lambda x. e\} \text{ in } \langle \text{ifdead } x_5 (f e') f, \pi_1 x_4 \rangle$$

If $x \notin FV(e)$ then type-inference would allow us to collect x_2 in this case (because then we can assign $x_5 : t \text{ weak}$ and $x_2 : t$), which would cause x_5 to be tombstoned. The problem is that the ifdead expression will always reduce to the dead case, which causes e' to be evaluated and then thrown away by f . Since by doing the type inference we already knew that the value of x_2 does not matter, we should be able to take the live branch and just pass a dummy value to f , which will throw it away.

The solution we propose to this problem is to add a new distinct tombstone marker d' . A weak reference that has been replaced with d' should be treated as alive for the purpose of ifdead reduction. A weak reference must only be tombstoned as d' if the value stored in the memory it weakly references is never used in the rest of the computation.

Formally, we extend the syntax of Hval

$$(\text{heap values}) hv ::= \dots \mid d'$$

and we change the ifdead reduction rule to be

$$\text{(ifdead)} \quad \text{letrec } H \text{ in } E[\text{ifdead } x e_1 e_2] \xrightarrow{\text{ifdead}} \begin{cases} \text{letrec } H \text{ in } E[e_2 w] & \text{if } H(x) = \text{weak } w \\ \text{letrec } H \text{ in } E[e_1] & \text{if } H(x) = d \\ \text{letrec } H \uplus \{z \mapsto 0\} \text{ in } E[e_2 z] & \text{if } H(x) = d' \end{cases}$$

We also use an additional typing rule, which assigns to d' the type $t \text{ weak}$ for a type variable t . We only need to allow d' to be typed by $t \text{ weak}$ because we will only introduce d' when tombstoning a weak reference to a binding that can be assigned a type variable. Observe that Theorem 4.4 still holds with this new rule for ifdead.

We can now prove the following theorem which states that given a program and a typing derivation that assigns some heap locations type variables, those locations can be rebound to 0 and weak references to those locations can be tombstoned with d' without affecting the result of the program.

Theorem 4.8 (Inference Weak GC). *Let*

$$\begin{aligned}
\Gamma_1 &= \{x_1 : t_1, \dots, x_n : t_n\}, \\
H_1^s &= \{x_1 \mapsto h_1, \dots, x_n \mapsto h_n\}, \\
H_1^w &= \{y_1 \mapsto \text{weak } x_{i_1}, \dots, y_m \mapsto \text{weak } x_{i_m}\}, \\
H_1^{t_1} &= \{x_1 \mapsto 0, \dots, x_n \mapsto 0\}, \\
H_1^{d'} &= \{y_1 \mapsto d', \dots, y_m \mapsto d'\}, \\
H_1 &= H_1^s \uplus H_1^w, \text{ and} \\
H'_1 &= H_1^{t_1} \uplus H_1^{d'}.
\end{aligned}$$

If

1. $\Gamma_1 \uplus \Gamma_2 \vdash e : \tau$ ($\tau \notin Tvar$), and
2. $\Gamma_1 \vdash H_2 : \Gamma_2$, and
3. $\exists S. \emptyset \vdash H_1 : S\Gamma_1$, and
4. $\text{letrec } H_1 \uplus H_2 \text{ in } e$ is well behaved

then $\text{letrec } H_1 \uplus H_2 \text{ in } e \equiv \text{letrec } H'_1 \uplus H_2 \text{ in } e$.

Proof sketch. Observe that any ifdead reduction step on some y_i which takes the live branch has the following form

$$\text{letrec } H \uplus \{y_i \mapsto \text{weak } x_k\} \text{ in } E[\text{ifdead } y_i e_1 e_2] \xrightarrow{\text{R-}\{\text{garb}\}} \text{letrec } H \uplus \{y_i \mapsto \text{weak } x_k\} \text{ in } e_2 x_k$$

If y_i had been tombstoned to d' we would have

$$\text{letrec } H \uplus \{y_i \mapsto d'\} \text{ in } E[\text{ifdead } y_i e_1 e_2] \xrightarrow{\text{R-}\{\text{garb}\}} \text{letrec } H \uplus \{y_i \mapsto d'\} \uplus \{z \mapsto 0\} \text{ in } e_2 z$$

Since x_k is assigned a type variable in Γ_1 , by Theorem 4.4 we can still assign it a type variable when typing $\text{letrec } H \uplus \{y_i \mapsto \text{weak } x_k\} \text{ in } e_2 x_k$, so we can replace the binding of x_k with 0 without affecting the reduction of the program. Therefore $\text{letrec } H_1 \uplus H_2 \text{ in } e \equiv \text{letrec } H'_1 \uplus H_2 \text{ in } e$. \square

5. Recovering Uniqueness of Program Result

In general, when a programmer uses weak references he or she does so in a way that guarantees that garbage collection cannot change the result of evaluation. Examples such as memoizing functions

and hash-consing lists certainly fit into this category. We refer to programs which always evaluate to the same result as *well-behaved* (note this is weaker than the usual notion of confluence). While the evaluation of these programs may not be deterministic, the final result is. An example of a program which we know will always have the same final result is

$$\text{letrec } \{\} \text{ in ifdead } (\text{weak } e) (e' e) e'$$

We can see that any end result will be the same as a result of the program $\text{letrec } H \text{ in } e' e$. Assuming there are no occurrences of $\text{weak } e''$ in e or e' for any e'' , all reductions of this program must end with the same result.

5.1 A Syntactic Restriction for Well-Behavedness

Given that it is impossible to syntactically characterize the well-behaved programs we will characterize a proper subset of the well-behaved programs which is big enough to cover some realistic uses of weak references. The syntactically restricted Exp^* is defined in Figure 5. The restriction comes from [5], also in that paper is an example implementation of (an approximation to) hash-consing meeting the restriction. This class is extremely restrictive as we essentially pair “ $\text{weak } e$ ” with e at each ifdead statement. While this hinders the usefulness of weak references, it does not destroy it. Weak references in this case are useful if the space required to store e is smaller than the data produced by e . This proper subset of well-behaved programs is referred to as the set of “gc-oblivious” programs.

Definition 5.1 (Companion Expressions). Let e_1 and e_2 be arbitrary expressions. We say that e_2 is the *companion* of e_1 if $(e_1, e_2) \in \text{ExpPair}$. (We do not use the relation “companion-of” symmetrically, i.e., e_1 is *not* the companion of e_2 .) \square

Definition 5.2 (GC-Oblivious Programs). A program $\text{letrec } \{\} \text{ in } e$ is *gc-oblivious* iff $e \in \text{Exp}^*$. \square

Proposition 5.3. *Membership in Exp^* is recognizable in linear time.*

Proof. The rules for ExpPair (and $\text{ExpPair}(p)$) are syntax directed by the first term of the pair and just walk down the terms comparing top-level constructors, except the weak case which uses syntactic equality. \square

Theorem 5.4. *If P is gc-oblivious then it is well-behaved.* \square

The proof of this theorem can be found at the end of this section.

Enlarging the Set of GC-Oblivious Programs

We can enlarge the set of gc-oblivious programs if we wish. For example, we can parameterize the ExpPair relation with $p \in \{1, 2\}^*$ to obtain a larger set of gc-oblivious programs. The parameter $p \in \{1, 2\}^*$ represents the sequence of projections that will yield an appropriate companion pair. $\text{ExpPair}(p)$ is defined as in Figure 6. We would then change the definition of Exp^* to use $\text{ExpPair}(p)$ in place of ExpPair .

We now give a proof of Theorem 5.4. We will begin by generalizing the notion of gc-obliviousness. This generalization of gc-obliviousness is used to show correctness of a semantics-preserving transformation which removes all occurrences of ifdead-expressions. Once ifdead-expressions are eliminated, well-behavedness of the calculus is easily proven using a “postponement” lemma along the lines of the proof given in [11].

Generalization of GC-Obliviousness

We will generalize the notion of gc-obliviousness by defining a set $\text{Exp}_{\text{Conf}} \supset \text{Exp}^*$ as in Figure 7.

We define Conf , a generalization of ExpPair , as follows:

Definition 5.5 (Relation Conf). For $e_1, e_2 \in \text{Exp}$. We define Conf by: $(e_1, e_2) \in \text{Conf}$ iff:

(The reduction of $\text{letrec } \{\} \text{ in } e_1$ gets stuck if and only if the reduction of $\text{letrec } \{\} \text{ in } e_2$ gets stuck) and the following hold for all reduction contexts E_1, E_2 :

1. $\text{letrec } \{\} \text{ in } E_1[e_1] \xrightarrow{R}^* \text{letrec } H' \uplus \{x \mapsto \text{weak } y\} \text{ in } E_1[x]$
iff $\text{letrec } \{\} \text{ in } E_2[e_2] \xrightarrow{R}^* \text{letrec } H' \text{ in } E_2[y]$.
2. $\text{letrec } \{\} \text{ in } E_1[e_1] \xrightarrow{R}^* \text{letrec } H' \uplus \{x \mapsto \lambda y. e'_1\} \text{ in } E_1[x]$
iff $\text{letrec } \{\} \text{ in } E_2[e_2] \xrightarrow{R}^* \text{letrec } H' \uplus \{x \mapsto \lambda y. e'_2\} \text{ in } E_2[x]$
such that for all $e \in \text{Exp}_{\text{Conf}}$ we have $(e_1 e, e_2 e) \in \text{Conf}$. \square

Note that this definition naturally generalizes when we expand the set of gc-oblivious programs. For example, if we added pairing and projection using the parameterized relation $\text{ExpPair}(p)$ for $p \in \{1, 2\}^*$, we could parameterize Conf as $\text{Conf}(p)$. $\text{Conf}(\varepsilon)$ would be defined by the above two clauses and $\text{Conf}(ip)$ would be defined by the above two clauses in addition to the following clause:

3. $\text{letrec } \{\} \text{ in } E_1[e_1] \xrightarrow{R}^* \text{letrec } H \uplus \{x \mapsto \langle y, z \rangle\} \text{ in } E_1[x]$
iff $\text{letrec } \{\} \text{ in } E_2[e_2] \xrightarrow{R}^* \text{letrec } H' \uplus \{x \mapsto \langle y, z \rangle\} \text{ in } E_2[x]$
such that $(\pi_i(e_1), \pi_i(e_2)) \in \text{Conf}(p)$.

In the following proof we will only use the first two clauses, but it is straightforward to extend the proof to encompass pairs and projections.

Lemma 5.6. *If $(e_1, e_2) \in \text{Conf}$ then $(e_1 \{x := e\}, e_2 \{x := e\}) \in \text{Conf}$*

Proof. Since no reduction rules branch on variables not in the heap, we can safely do the substitution on a reduction sequence and still have a valid reduction sequence. \square

Lemma 5.7. *If $(e_1 \{x := e\}, e_2 \{x := e\}) \in \text{Conf}$ then $((\lambda x. e_1)e, (\lambda x. e_2)e) \in \text{Conf}$*

Proof. If $\text{letrec } \{\} \text{ in } e$ diverges or gets stuck then

$$((\lambda x. e_1)e, (\lambda x. e_2)e) \in \text{Conf}$$

vacuously. The only other possibility is that

$$\text{letrec } \{\} \text{ in } e \xrightarrow{R}^* \text{letrec } H' \text{ in } y$$

(pick H' such that $FV(e_1) \cap FV(H') = \emptyset$) in which case we can use a redex labeling argument to show

$$\begin{aligned} &\text{letrec } \{\} \text{ in } e_1 \{x := e\} \xrightarrow{R}^* \text{letrec } H'' \text{ in } z \\ \text{iff } &\text{letrec } H' \text{ in } e_1 \{x := y\} \xrightarrow{R}^* \text{letrec } H''' \text{ in } z \end{aligned}$$

such that $\text{result}(H'', z) = \text{result}(H''', z)$. \square

Lemma 5.8. *$(e_1, e_2) \in \text{ExpPair}$ implies $(e_1, e_2) \in \text{Conf}$*

Proof. By induction on the derivation of $(e_1, e_2) \in \text{ExpPair}$

$$\text{case : } \frac{e \in \text{Exp}^*}{(\text{weak } e, e) \in \text{ExpPair}}$$

It is clear both cases of the definition of Conf hold (the second case holds vacuously) and that e gets stuck iff $\text{weak } e$ gets stuck.

$$\text{case : } \frac{(e_1, e_2) \in \text{ExpPair}}{(\lambda x. e_1, \lambda x. e_2) \in \text{ExpPair}}$$

Neither side of the pair can get stuck because they both step to a value immediately. The first case of Conf is vacuous. By IH and Lemma 5.6 we have $(e_1 \{x := e\}, e_2 \{x := e\}) \in \text{Conf}$, so by

$$\begin{array}{c}
\frac{}{i \in \text{Exp}^*} \quad \frac{}{x \in \text{Exp}^*} \quad \frac{e_1, e_2 \in \text{Exp}^*}{\langle e_1, e_2 \rangle \in \text{Exp}^*} \quad \frac{e \in \text{Exp}^* \quad i \in \{1, 2\}}{\pi_i e \in \text{Exp}^*} \quad \frac{e \in \text{Exp}^*}{\lambda x. e \in \text{Exp}^*} \quad \frac{e_1, e_2 \in \text{Exp}^*}{e_1 e_2 \in \text{Exp}^*} \\
\frac{e \in \text{Exp}^*}{\text{weak } e \in \text{Exp}^*} \quad \frac{(e_1, e_2) \in \text{ExpPair} \quad e_3 \in \text{Exp}^*}{\text{ifdead } e_1 (e_3 e_2) e_3 \in \text{Exp}^*} \quad \frac{e \in \text{Exp}^*}{(\text{weak } e, e) \in \text{ExpPair}} \quad \frac{(e_1, e_2) \in \text{ExpPair}}{(\lambda x. e_1, \lambda x. e_2) \in \text{ExpPair}} \\
\frac{(e_1, e_2) \in \text{ExpPair} \quad e_3 \in \text{Exp}^*}{(e_1 e_3, e_2 e_3) \in \text{ExpPair}} \quad \frac{(e_1, e_2) \in \text{ExpPair} \quad (e_3, e_4) \in \text{ExpPair}}{(\text{ifdead } e_1 (e_3 e_2) e_3, \text{ifdead } e_1 (e_4 e_2) e_4) \in \text{ExpPair}}
\end{array}$$

Figure 5. Definition of Exp^*

$$\begin{array}{c}
\frac{e \in \text{Exp}^*}{(\text{weak } e, e) \in \text{ExpPair}(\varepsilon)} \quad \frac{(e_1, e_2) \in \text{ExpPair}(ip) \quad i \in \{1, 2\}}{(\pi_i e_1, \pi_i e_2) \in \text{ExpPair}(p)} \\
\frac{(e_1, e_2) \in \text{ExpPair}(p) \quad e_3 \in \text{Exp}^*}{((e_1, e_3), (e_2, e_3)) \in \text{ExpPair}(1p)} \quad \frac{(e_1, e_2) \in \text{ExpPair}(p) \quad e_3 \in \text{Exp}^*}{((e_3, e_1), (e_3, e_1)) \in \text{ExpPair}(2p)} \\
\frac{(e_1, e_2) \in \text{ExpPair}(p)}{(\lambda x. e_1, \lambda x. e_2) \in \text{ExpPair}(p)} \quad \frac{(e_1, e_2) \in \text{ExpPair}(p) \quad e_3 \in \text{Exp}^*}{(e_1 e_3, e_2 e_3) \in \text{ExpPair}(p)} \\
\frac{(e_1, e_2) \in \text{ExpPair}(\varepsilon) \quad (e_3, e_4) \in \text{ExpPair}(p)}{(\text{ifdead } e_1 (e_3 e_2) e_3, \text{ifdead } e_1 (e_4 e_2) e_4) \in \text{ExpPair}(p)}
\end{array}$$

Figure 6. Extended $\text{ExpPair}(p)$ Definition

$$\begin{array}{c}
\frac{}{i \in \text{Exp}_{\text{Conf}}} \quad \frac{}{x \in \text{Exp}_{\text{Conf}}} \quad \frac{e_1, e_2 \in \text{Exp}_{\text{Conf}}}{\langle e_1, e_2 \rangle \in \text{Exp}_{\text{Conf}}} \quad \frac{e \in \text{Exp}_{\text{Conf}} \quad i \in \{1, 2\}}{\pi_i e \in \text{Exp}_{\text{Conf}}} \\
\frac{e \in \text{Exp}_{\text{Conf}}}{\lambda x. e \in \text{Exp}_{\text{Conf}}} \quad \frac{e_1, e_2 \in \text{Exp}_{\text{Conf}}}{e_1 e_2 \in \text{Exp}_{\text{Conf}}} \quad \frac{e \in \text{Exp}_{\text{Conf}}}{\text{weak } e \in \text{Exp}_{\text{Conf}}} \quad \frac{(e_1, e_2) \in \text{Conf} \quad e_3 \in \text{Exp}_{\text{Conf}}}{\text{ifdead } e_1 (e_3 e_2) e_3 \in \text{Exp}_{\text{Conf}}}
\end{array}$$

Figure 7. Definition of Exp_{Conf}

Lemma 5.7 we have $((\lambda x. e_1)e, (\lambda x. e_2)e) \in \text{Conf}$, so the second case of Conf holds

$$\text{case : } \frac{(e_1, e_2) \in \text{ExpPair} \quad e_3 \in \text{Exp}^*}{(e_1 e_3, e_2 e_3) \in \text{ExpPair}}$$

By IH we have

$$\begin{array}{l}
\text{letrec } \{\} \text{ in } E'_1[e_1] \xrightarrow{R^*} \text{letrec } H'' \uplus \{x \mapsto \lambda y. e'_1\} \text{ in } E'_1[x] \\
\text{iff letrec } \{\} \text{ in } E'_2[e_2] \xrightarrow{R^*} \text{letrec } H'' \uplus \{x \mapsto \lambda y. e'_2\} \text{ in } E'_2[x]
\end{array}$$

such that for all $e : \text{Exp}_{\text{Conf}}$ we have $(e_1 e, e_2 e) \in \text{Conf}$, so in particular $(e_1 e_3, e_2 e_3) \in \text{Conf}$.

$$\text{case : } \frac{(e_1, e_2) \in \text{ExpPair} \quad (e_3, e_4) \in \text{ExpPair}}{(\text{ifdead } e_1 (e_3 e_2) e_3, \text{ifdead } e_1 (e_4 e_2) e_4) \in \text{ExpPair}}$$

Suppose $\text{letrec } \{\} \text{ in ifdead } e_1 (e_3 e_2) e_3 \xrightarrow{R^*} \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } x$. Since reduction did not get stuck it must be that

$$\begin{array}{l}
\text{letrec } H \text{ in } e_1 \xrightarrow{R^*} \text{letrec } H_1 \uplus \{x_1 \mapsto \text{weak } y\} \text{ in } x_1 \\
\text{and letrec } H \text{ in } e_3 \xrightarrow{R^*} \text{letrec } H_2 \uplus \{x_2 \mapsto \lambda y. e'_3\} \text{ in } x_2 \\
\text{and letrec } H_1 \uplus H_2 \text{ in } x_2 y \xrightarrow{R^*} \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } x.
\end{array}$$

By IH we have $\text{letrec } \{\} \text{ in } e_2 \xrightarrow{R^*} \text{letrec } H_1 \text{ in } y$ and $\text{letrec } \{\} \text{ in } e_4 \xrightarrow{R^*} \text{letrec } H_2 \uplus \{x_2 \mapsto \lambda y. e'_4\} \text{ in } x_2$ and for $e : \text{Exp}^*$ we have $(e_3 e_2, e_4 e_2) \in \text{Conf}$. Putting this together we have

$$\text{ifdead } e_1 (e_4 e_2) e_4 \xrightarrow{R^*} \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } x.$$

The reverse direction is similar.

Suppose $\text{letrec } \{\} \text{ in ifdead } e_1 (e_3 e_2) e_3$ gets stuck. If it gets stuck during evaluation of e_1 then $\text{ifdead } e_1 (e_4 e_2) e_4$ gets stuck there also. If it gets stuck during evaluation of $(e_3 e_2)$ then by IH $\text{ifdead } e_1 (e_4 e_2) e_4$ also gets stuck there. Similarly if it gets stuck during evaluation of $(e_3 y)$ (after reducing e_1 to y), then by IH $\text{ifdead } e_1 (e_4 e_2) e_4$ gets stuck in the same place. The reverse direction is similar. \square

We define the transformation e° as follows:

$$\begin{array}{l}
x^\circ = x \\
i^\circ = i \\
\langle e_1, e_2 \rangle^\circ = \langle e_1^\circ, e_2^\circ \rangle \\
(\pi_i(e_1))^\circ = \pi_i(e_1^\circ) \\
(e_1 e_2)^\circ = e_1^\circ e_2^\circ \\
(\text{weak } e_1)^\circ = \text{weak } (e_1^\circ) \\
(\text{ifdead } e_1 (e_2 e_3) e_3)^\circ = (e_1^\circ e_2^\circ e_3^\circ)
\end{array}$$

Lemma 5.9. Suppose $e_0 \in \text{Exp}_{\text{Conf}}$ then one of the following holds with regards to the reduction of $\text{letrec } \{\} \text{ in } e_0$.

1. $\text{letrec } \{\} \text{ in } e_0$ always gets stuck.
2. $\text{letrec } \{\} \text{ in } e_0 \xrightarrow{R^*} \text{letrec } H'' \text{ in } x$ if and only if $\text{letrec } \{\} \text{ in } e_0^\circ \xrightarrow{R^*} \text{letrec } H''' \text{ in } x$ with $\text{result}(H'', x) = \text{result}(H''', x)$ and $\text{letrec } \{\} \text{ in } e_0$ gets stuck if and only if $\text{letrec } \{\} \text{ in } e_0^\circ$ gets stuck.

Proof. By structural induction. The only interesting case is $e_0 = \text{ifdead } e_1 (e e_2) e$ because the IH carries through immediately in all other cases.

The first thing to be evaluated is e_1 , and one of the following must hold of this evaluation by IH.

1. $\text{letrec } \{ \}$ in e_1 always gets stuck
2. $\text{letrec } \{ \}$ in $e_1 \xrightarrow{R,*} \text{letrec } H$ in x if and only if $\text{letrec } \{ \}$ in $e_1^\circ \xrightarrow{R,*} \text{letrec } H'$ in x with $\text{result}(H, x) = \text{result}(H', x)$ and $\text{letrec } \{ \}$ in e_1 gets stuck if and only if $\text{letrec } \{ \}$ in e_1° gets stuck.

In the first case $\text{letrec } \{ \}$ in $\text{ifdead } e_1 (e e_2) e$ always gets stuck.

In the second case, since e_1° has no occurrences of ifdead -expressions, its evaluation is deterministic (modulo garbage collection) and one of the following hold.

1. There is no reduction sequence $\text{letrec } \{ \}$ in $e_1 \xrightarrow{R,*} \text{letrec } H' \uplus \{x \mapsto \text{weak } y\}$ in x so consequently

$$\text{letrec } \{ \} \text{ in } \text{ifdead } e_1 (e e_2) e$$

always gets stuck or else reduces to

$$\text{letrec } H' \uplus \{x \mapsto \text{weak } y\} \text{ in } (e e_2).$$

Since $(e e_2)$ is unevaluated

$$\begin{aligned} & \text{letrec } H' \uplus \{x \mapsto \text{weak } y\} \text{ in } (e e_2) \xrightarrow{R,*} \text{letrec } H'' \text{ in } e'' \\ \text{iff } & \text{letrec } \{ \} \text{ in } (e e_2) \xrightarrow{R,*} \text{letrec } H''' \text{ in } e''' \end{aligned}$$

with $\text{result}(H'', e'') = \text{result}(H''', e''')$, therefore we can use the IH on $\text{letrec } \{ \}$ in $e e_2$ to finish.

2. Since $\text{letrec } \{ \}$ in $\text{ifdead } e_1 (e e_2) e \in \text{Prog}^*$, we have $\text{letrec } \{ \}$ in $e_1 \xrightarrow{R,*} \text{letrec } H' \uplus \{x \mapsto \text{weak } y\}$ in x if and only if $\text{letrec } \{ \}$ in $e_2 \xrightarrow{R,*} \text{letrec } H'$ in x by Lemma 5.8. From here it is clear that we have

$$\text{letrec } \{ \} \text{ in } \text{ifdead } e_1 (e e_2) e \xrightarrow{R,*} \text{letrec } H'' \text{ in } x$$

if and only if $\text{letrec } \{ \}$ in $e e_2 \xrightarrow{R,*} \text{letrec } H'''$ in x with $\text{result}(H'', x) = \text{result}(H''', x)$ by using the IH after reducing the outermost ifdead as in the previous case. \square

Proof of Theorem 5.4. From Lemma 5.9 we know that $\text{letrec } \{ \}$ in e always gets stuck, or it yields the same result as the evaluation of $\text{letrec } \{ \}$ in e° . Since e° contains no ifdead -expressions, the evaluation of $\text{letrec } \{ \}$ in e° is deterministic, therefore the evaluation of $\text{letrec } \{ \}$ in e is deterministic. \square

Corollary 5.10. *For any well-typed, gc-oblivious program $P = \text{letrec } H$ in e , we have $P \equiv \text{letrec } H$ in e° .*

Proof. No well-typed program can evaluate to a stuck term, so Theorem 5.4 gives us the result. \square

5.2 A General Semantic Criterion for Well-Behavedness

The syntactic restriction given previously is arguably too restrictive and does not allow natural expression of many realistic uses of weak references. In particular memoized functions do not seem to fall into this restricted category. In order to remedy this situation we give a natural semantic criterion for well-behavedness and use this criterion to informally argue for the correctness of an implementation of memoized function application. This semantic criterion is intended to be used by programmers to assure their programs written using weak references are well-behaved (which is generally desirable). We assume that we are working within a typed setting so

that we do not have to worry about stuck programs and to provide a free variable context for ifdead expressions under lambda binders.

Local Well-behavedness

We say a program $\text{letrec } H$ in e is locally well-behaved if it is well-behaved at each ifdead . In order to define this we use the following relation.

Definition 5.11 (Loc_Γ). $(e_1, e_2, e_3) \in \text{Loc}_\Gamma$ iff for all H such that $\vdash H : \Gamma$, if

$$\text{letrec } H \text{ in } e_1 \xrightarrow{R,*} \text{letrec } H' \uplus \{x \mapsto \text{weak } y, y \mapsto hv\} \text{ in } x$$

then we have

$$(\text{letrec } H' \uplus \{x \mapsto \text{weak } y, y \mapsto hv\} \text{ in } e_3 \xrightarrow{R,*} \text{letrec } H'' \text{ in } z \text{ iff}$$

$$\text{letrec } H' \uplus \{x \mapsto \text{weak } y, y \mapsto hv\} \text{ in } e_2 \xrightarrow{R,*} \text{letrec } H''' \text{ in } z)$$

$$\text{with } \text{result}(H'', z) = \text{result}(H''', z). \quad \square$$

Definition 5.12. A closed program $\text{letrec } H$ in e is *locally well-behaved* iff it has a typing derivation $\vdash \text{letrec } H \text{ in } e : \tau$ and for all ifdead occurrences in the derivation, $\Gamma \vdash \text{ifdead } e_1 e_2 e_3 : \tau'$, we have $(e_1, e_2, e_3) \in \text{Loc}_\Gamma$. \square

This notion of local well-behavedness is decidable, though in an unfeasably long time, but only because our core language is essentially simply-typed lambda calculus. The addition of a fixpoint operator would make this undecidable. However this is still a natural criterion to use when programming with weak references.

Theorem 5.13. *If a program P is gc-oblivious and well-typed, then it is locally well-behaved.*

Proof sketch. Every occurrence of an ifdead in a gc-oblivious program obviously satisfies the second part of the property (same results of reducing each branch) all that is missing is well-typedness. \square

Theorem 5.14. *If a program P is locally well-behaved then it is well-behaved.*

Proof sketch. Since the program is well-behaved around each of its top-level ifdead occurrences and ifdead reduction is the only source non-well-behavedness, it is well-behaved. \square

EXAMPLE 5.15 (MEMOIZING FUNCTIONS). Assume we have a type $\text{memofun}(\tau_1, \tau_2)$ of memoized functions from τ_1 to τ_2 with the following functions:

$$\begin{aligned} \text{Lookupmemo} & : (\text{memofun}(\tau_1, \tau_2) * \tau_1) \rightarrow \tau_2 \text{ weak option} \\ \text{Addmemo} & : (\text{memofun}(\tau_1, \tau_2) * \tau_1) \rightarrow \\ & \quad (\tau_2 * \text{memofun}(\tau_1, \tau_2)) \end{aligned}$$

Lookupmemo and Addmemo do not need to use ifdead so they are locally well-behaved. We try to verify the following (in ML-like notation) is locally well-behaved:

```
fun appmemo (f:memofun(T1,T2),o:T1)
  :(T2 * memofun(T1,T2)) =
  case Lookupmemo(f,o) of
  None => Addmemo(f,o)
  | Some(ref) =>
    (ifdead (ref) (Addmemo(f,o)) (fn x => (x,f)))
```

Intuitively this should fit our definition of locally well-behaved. Formally we need to prove something about the semantics of the ifdead expression for all H such that $\text{ref}, \text{Lookupmemo}$ and Addmemo have the appropriate types. This is not possible because

we rely on the dynamic semantics of *Addmemo* to make the argument, not merely its type. So we assume that *Addmemo* is inlined. Then we need $\text{letrec } H \text{ in } ((\lambda x. \langle x, f \rangle) y) \xrightarrow{R}^* \text{letrec } H' \text{ in } z$ iff $\text{letrec } H \text{ in } \text{Addmemo}(f, o) \xrightarrow{R}^* \text{letrec } H'' \text{ in } z$ with $\text{result}(H', z) = \text{result}(H'', z)$. If *ref* is dead then

$$\text{letrec } H \text{ in } \text{Addmemo}(f, o)$$

re-adds the corresponding dead entry, if *ref* is still alive then $\text{letrec } H \text{ in } ((\lambda x. \langle x, f \rangle) y)$ still has the corresponding entry and returns the same pair that $\text{letrec } H \text{ in } \text{Addmemo}(f, o)$ does. So this example is locally well-behaved, so it is well-behaved. \square

6. Related Work

Morrisett, Felleisen and Harper's [11] was the first work to fully specify the static and dynamic semantics of a language with garbage collection and was followed up by [12], which further develops the theory in the context of a polymorphic language.

Most of the work specifically related to weak references is in actual implementations of programming languages and in their informal descriptions. Aside from that, Peyton Jones, Marlow, and Elliot use weak references (and some other Haskell features) to implement memoized functions in [8]. Marizen, Zendra and Colnet discuss the addition of weak and soft references to the Eiffel language and the advantages parametric polymorphism in this context [9]. Neither of these papers contain formal semantics and we are aware of no other attempt to formalize the semantics of weak references.

7. Conclusion and Future Work

In this paper we introduce and investigate a formal semantics of a functional language with weak references. We show the flexibility of the semantic framework by extending it to the case of the key/value weak references found in Haskell. We extend type-inference based reachable garbage collection to allow collection of additional weak references without incurring computational overhead to recompute data stored in them. We address the well-behaved usage of weak references by proving that a syntactically restricted set of programs has a unique program result, regardless of garbage collection.

The method of proof is of independent interest. We use a relation to prove the correctness of a transformation which removes all ifdead expressions. Such a transformation is fairly easy for a programmer to do in mind in order to see what the final outcome of the program will be. In addition, we have provided a general semantic criterion which can be used by programmers to create well-behaved programs without having to adhere to a strict syntactic discipline.

In the future we hope to be able to use this formal semantics for weak references to investigate more complex languages which combine weak references with other programming features, such as reference mutation and finalization. It may also be fruitful to investigate denotational semantics for a language with weak references.

Acknowledgments

We would like to thank Joe Wells for the interactions in which he offered great simplifications and pointed out several initial flaws. Greg Morrisett carefully read an early draft and gave us many beneficial suggestions for improvement. Franklyn Turbak was very helpful in getting this work off the ground. He was able to point us in the right direction initially, for which we thank him. Adam Bakewell, Sebastien Carlier, Chiyan Chen, Likai Lui, Peter Moller Neergaard, and Hongwei Xi all spent time talking with the authors and offered valuable suggestions and opinions.

References

- [1] APPEL, A. W. Runtime tags aren't necessary. *Lisp and Symbolic Computation* 2, 2 (1989), 153–162.
- [2] BAKER, H. G. Unify and conquer. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming* (New York, NY, USA, 1990), ACM Press, pp. 218–226.
- [3] CHAILLOUX, E., MANOURY, P., AND PAGANO, B. *Developing Applications with Objective Caml*. O'Reilly, France, 2000.
- [4] DONNELLY, K., AND KFOURY, A. J. Some considerations on formal semantics for weak references. Technical Report Don+Kfo:BUCS-TR-2005-X, Department of Computer Science, Boston University, July 2005. <http://types.bu.edu/reports/Don+Kfo:BUCS-TR-2005-X.html>.
- [5] HALLETT, J. J., AND KFOURY, A. J. A formal semantics for weak references. Technical Report Hal+Kfo:BUCS-TR-2005-X, Department of Computer Science, Boston University, May 2005. <http://types.bu.edu/reports/Hal+Kfo:BUCS-TR-2005-X.html>.
- [6] HAYES, B. Ephemeron: a new finalization mechanism. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1997), ACM Press, pp. 176–183.
- [7] THE HUGS/GHC TEAM. *Hugs/GHC Extension Libraries: Weak*. <http://www.dcs.gla.ac.uk/fp/software/ghc/lib/hg-libs-15.html>.
- [8] JONES, S. L. P., MARLOW, S., AND ELLIOTT, C. Stretching the storage manager: Weak pointers and stable names in Haskell. In *IFL '99: Selected Papers from the 11th International Workshop on Implementation of Functional Languages* (London, UK, 2000), Springer-Verlag, pp. 37–58.
- [9] MERIZEN, F., ZENDRA, O., AND COLNET, D. Designing efficient and safe non-strong references in Eiffel with parametric types. Research Report A04-R-149, INRIA Lorraine / LORIA UMR 7503, Sep 2004.
- [10] MORRISSETT, G., 2005. private communication.
- [11] MORRISSETT, G., FELLEISEN, M., AND HARPER, R. Abstract models of memory management. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture* (New York, NY, USA, 1995), ACM Press, pp. 66–77.
- [12] MORRISSETT, G., AND HARPER, R. Semantics of memory management for polymorphic languages. In *Higer Order Operational Techniques in Semantics*, A. Gordon and A. Pitts, Eds. Newton Institute, Cambridge University Press, 1997.
- [13] THE SMLofNJ TEAM. *SML of NJ Structure Documentation: The WEAK signature*. <http://www.smlnj.org/doc/SMLofNJ/pages/weak.html>.
- [14] SUN MICROSYSTEMS. *Java 2 Platform SE v1.3.1: package java.lang.ref*. <http://java.sun.com/j2se/1.3/docs/api/java/lang/ref/package-summary.html>.
- [15] TOLMACH, A. P. Tag-free garbage collection using explicit type parameters. In *LISP and Functional Programming* (1994), pp. 1–11.
- [16] WRIGHT, A. K., AND FELLEISEN, M. A syntactic approach to type soundness. *Information and Computation* 115, 1 (1994), 38–94.