

Some Considerations on a Calculus with Weak References

Kevin Donnelly
Boston University
kevind@cs.bu.edu

Assaf J. Kfoury
Boston University
kfoury@cs.bu.edu

July 6, 2005

Abstract

Weak references are references that do not prevent the object they point to from being garbage collected. Most realistic languages, including Java, SML/NJ, and OCaml to name a few, have some facility for programming with weak references. Weak references are used in implementing idioms like memoizing functions and hash-consing in order to avoid potential memory leaks.

However, the semantics of weak references in many languages are not clearly specified. Without a formal semantics for weak references it becomes impossible to prove the correctness of implementations making use of this feature. Previous work by Hallett and Kfoury [HK05] extends λ_{gc} [MFH95], a language for modeling garbage collection, to λ_{weak} , a similar language with weak references.

Using this previously formalized semantics for weak references, we consider two issues related to well-behavedness of programs. Firstly, we provide a new, simpler proof of the well-behavedness of the syntactically restricted fragment of λ_{weak} defined in [HK05]. Secondly, we give a natural semantic criterion for well-behavedness much broader than the syntactic restriction, which is useful as principle for programming with weak references.

Furthermore we extend the result, proved in [MFH95], which allows one to use type-inference to collect some reachable objects that are never used. We prove that this result holds of our language, and we extend this result to allow the collection of weakly-referenced reachable garbage without incurring the computation overhead sometimes associated with collecting weak bindings (e.g. the need to recompute a memoized function).

Lastly we use extend the semantic framework to model the key/value weak references found in Haskell and we prove the Haskell is semantics equivalent to a simpler semantics due to the lack of side-effects in our language.

1 Introduction

The ability to concisely specify and formally prove the correctness of garbage collection strategies was an important contribution of Morrisett, Felleisen and Harper's λ_{gc} . 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. However, this does not negate the usefulness of having a high-level formal model which specifies how garbage collection and weak references interact. In fact, some (perhaps informal) model is critical to writing correct implementations using weak references.

A formal model, which extends λ_{gc} with the means to introduce and conditionally dereference weak references, is introduced in [HK05]. In this language, as in λ_{gc} , all values are allocated to, and stay on the heap during evaluation (unless garbage collected). For this reason it makes sense to create a weak reference to any program value. The semantics given essentially matches the semantics for weak references

in SML/NJ [SML]. Because 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, this is a bit misleading. The semantics used in this paper is not ambiguous and if an SML programmer uses this semantics his programs will behave correctly as long as their correctness does not depend on weak references having been collected. This is a natural restriction as the programmer (usually) has no control over if or when garbage collection occurs. Any usage of weak references to avoid memory leaks should fit this criterion.

The contributions of this paper are several. Firstly, we provide a much simpler and complete proof of the well-behavedness of a syntactically restricted class of programs. The original proof was quite complicated and only proved a part of the theorem. Secondly, we provide a natural semantic criterion for well-behavedness which is much broader than the syntactic restriction. This criterion is a useful principle for programming with weak reference, and we use it to argue for the correctness of an implementation of memoizing function application. Thirdly, we extend the result from [MFH95] which allows the use of type-inference to collect some objects which are reachable but never used. The extended result additionally allows tombstoning of weak references to this reachable garbage, without incurring the runtime overhead sometimes associated with tombstoning a weak reference. Lastly, we show the flexibility of the framework by extending it to formalize a model of the key/value weak references found in Haskell.

The paper is organized as follows. In Section 2 we introduce the syntax and semantics of λ_{weak} . In Section 3 we explain the syntactic restriction that we impose to assure uniqueness of program result. In Section 4 we prove that restricted programs have a unique result. In Section 5 we give a natural semantic criterion for well-behavedness. In Section 6 we prove the correctness of a transformation that uses type inference to both do extra garbage collection, and tombstone weak references without requiring the extra computation that might normally be necessary. In Section 7 we formalize the semantics of the key/value weak references found in Haskell and prove it equivalent to a simpler semantics in the absence of side-effects.

2 Modeling weak references: λ_{weak}

In this section we give the formal syntax and semantics of λ_{weak} .

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 ((e_1, e_2)), 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**. 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 $H : \text{Var} \rightarrow \text{Hval}$) 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 $\text{Dom}(H)$ to be the domain of H and we define $H^s = \{(x, H(x)) \mid H(x) \neq \text{weak } y \text{ for any } y\}$ 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

Programs:			
(variables)	$w, x, y, z \in \text{Var}$		
(integers)	$i \in \text{Int}$	$::=$	$\dots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \dots$
(expressions)	$e \in \text{Exp}$	$::=$	$x \mid i \mid \langle e_1, e_2 \rangle \mid \pi_i e \mid \pi_2 e \mid \lambda x.e \mid e_1 e_2 \mid$ $\text{weak } e \mid \text{ifdead } e_1 e_2 e_3$
(heap values)	$hv \in \text{Hval}$	$::=$	$i \mid \langle x_1, x_2 \rangle \mid \lambda x.e \mid \text{weak } x \mid \mathbf{d}$
(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)	$E \in \text{Ctx}$	$::=$	$[] \mid \langle E, e \rangle \mid \langle x, E \rangle \mid \pi_i E \mid E e \mid x E \mid \text{weak } E \mid \text{ifdead } E e_1 e_2$
(instruction)	$I \in \text{Instr}$	$::=$	$hv \mid \pi_i x \mid x y \mid \text{ifdead } x e_1 e_2$
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}

heap H and a program $\text{letrec } H \text{ in } e$ are defined by:

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

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

Expressions are identified up to α -conversion and programs are identified up to renaming of variables bound in the heap ,e.g., $\text{letrec } H \uplus \{x \mapsto h\} \text{ in } x = \text{letrec } H \uplus \{y \mapsto h\} \text{ in } y$ 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 $P = \text{letrec } H \text{ in } E[\text{ifdead } x e_2 e_3]$ does a conditional

EXAMPLE 2.1.

$$\begin{array}{l}
\text{letrec } \{ \} \text{ in } (\lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y)) \langle 5, 6 \rangle \\
\begin{array}{l} \xrightarrow{\text{alloc}} \\ \xrightarrow{\text{alloc}} \\ \xrightarrow{\text{alloc}} \\ \xrightarrow{\text{alloc}} \end{array} \text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y) \} \text{ in } a \langle 5, 6 \rangle \\
\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 \\
\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 \\
\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 \\
\begin{array}{l} \xrightarrow{\text{app}} \\ \xrightarrow{\text{alloc}} \end{array} \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) \\
\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) \\
\begin{array}{l} \xrightarrow{\text{ifdead}} \\ \longrightarrow \dots \xrightarrow{\text{garb}} \end{array} \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 \\
\text{or} \\
\begin{array}{l} \xrightarrow{\text{garb}} \\ \longrightarrow \dots \xrightarrow{\text{garb}} \end{array} \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
\end{array}$$

where a, b, c, e, f and g are fresh variables introduced in the process of program evaluation. □

Figure 2: Example of Non-confluent Reduction

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 $H(x) = d$ 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.

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

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

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

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 garbage. Often, in practice garbage collection will collect all locations to which there is no 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).

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 shown in Figure 2, taken from [HK05], shows the non-confluence of λ_{weak} .

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

Figure 3: Definition of $\text{result}(H, e)$

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

Figure 4: Definition of Exp^*

3 Restoring Confluence

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. In order to formally state what we mean by the same result, we define $\text{result}(H, E)$ as in Figure 3.

The following definitions capture the notion of a program having a unique result.

Definition 3.1 (Eval-Set). The evaluation set of a program P relative to rewrite rules G :

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

We write $\text{eval-set}(P)$ for $\text{eval-set}(P, R)$ □

Definition 3.2 (Program Equivalence). $(P, G) \equiv (P', G')$ iff $eval\text{-}set(P, G) = eval\text{-}set(P', G')$. If $G = G' = R$, we simply write $P \equiv P'$. \square

Definition 3.3 (Well-Behaved Programs). A program P is well-behaved iff $eval\text{-}set(P)$ is a singleton. \square

A program is well-behaved if it has a unique result of evaluation. In general it is undecidable whether a given program is well-behaved because we can reduce non-termination in untyped lambda calculus (which is undecidable) to this property.

Proposition 3.4. *For an arbitrary program P it is undecidable if P is well-behaved.*

Proof. Let e be an untyped lambda term with $x \notin FV(e)$, $\Omega = \omega \omega$ and $\omega = (\lambda x. x x)$. Then

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

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

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 many realistic uses of weak references. The syntactically restricted Exp^* is defined in Figure 4. The restriction comes from [HK05], also in that paper is an example implementation of hash-consing meeting the restriction. This proper subset of well-behaved programs is referred to as the set of “gc-oblivious” programs.

Definition 3.5 (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 3.6 (GC-Oblivious Programs). A program $\text{letrec } \{ \} \text{ in } e$ is *gc-oblivious* iff $e \in \text{Exp}^*$. \square

Theorem 3.7 (GC-Oblivious Programs Are Well-Behaved). *If P is gc-oblivious then it is well-behaved.* \square

The proof of this theorem can be found in Section 4.

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 5. We would then change the definition of Exp^* to use $\text{ExpPair}(p)$ in place of ExpPair .

4 Proof of GC-Oblivious Well-Behavedness

In this section we give an alternative proof of Theorem 3.7, the third part of which is proved in [HK05] in a less straightforward manner. 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 [MFH95].

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

Figure 5: Extended $\text{ExpPair}(p)$ Definition

Generalization of GC-Obliviousness

We will generalize the notion of GC-Obliviousness by defining a set $\text{Exp}_{\text{Conf}} \supset \text{Exp}^*$.

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

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

Definition 4.1 (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 $\text{letrec } \{ \}$ in e_1 gets stuck if and only if the reduction of $\text{letrec } \{ \}$ in e_2 gets stuck and the following hold for all reduction contexts E_1, E_2 :

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

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. $\text{letrec } \{ \}$ in $E_1[e_1] \xrightarrow{\text{R}}^* \text{letrec } H \uplus \{x \mapsto \langle y, z \rangle\}$ in $E_1[x]$
iff $\text{letrec } \{ \}$ in $E_2[e_2] \xrightarrow{\text{R}}^* \text{letrec } H' \uplus \{x \mapsto \langle y, z \rangle\}$ 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 4.2. *If $(e_1, e_2) \in \text{ExpPair}$ and $\text{letrec } \{ \}$ in $E[e_1] \xrightarrow{\text{R}}^* \text{letrec } H'$ in $E[x]$ then either $H'(x) = \text{weak } y$ or $H'(x) = \lambda y. e$.*

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

Lemma 4.3. *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 4.4. *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 $\text{letrec } \{ \}$ 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 $\text{letrec } \{ \}$ in $e \xrightarrow{R^*} \text{letrec } H'$ 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 $\text{letrec } \{ \}$ in $e_1\{x := e\} \xrightarrow{R^*} \text{letrec } H''$ in z iff $\text{letrec } H'$ in $e_1\{x := y\} \xrightarrow{R^*} \text{letrec } H'''$ in z such that $\text{result}(H'', z) = \text{result}(H''', z)$. \square

Lemma 4.5. *$(e_1, e_2) \in \text{ExpPair}$ 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 4.3 we have $(e_1\{x := e\}, e_2\{x := e\}) \in \text{Conf}$, so by Lemma 4.4 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

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

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

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

By IH we have $\text{letrec } \{ \}$ in $e_2 \xrightarrow{R^*} \text{letrec } H_1$ in y and $\text{letrec } \{ \}$ in $e_4 \xrightarrow{R^*} \text{letrec } H_2 \uplus \{x_2 \mapsto \lambda y. e'_4\}$ 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 $\text{letrec } \{\}$ in $\text{ifdead } e_1 (e_3 e_2) e_3$ 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:

$$\begin{aligned} x^\circ &= x \\ i^\circ &= i \\ \langle e_1, e_2 \rangle^\circ &= \langle e_1^\circ, e_2^\circ \rangle \\ (\pi_i(e_1))^\circ &= \pi_i(e_1^\circ) \\ (e_1 e_2)^\circ &= e_1^\circ e_2^\circ \\ (\mathbf{weak } e_1)^\circ &= \mathbf{weak } (e_1^\circ) \\ (\text{ifdead } e_1 (e e_2) e)^\circ &= (e^\circ e_2^\circ) \end{aligned}$$

Lemma 4.6. *Suppose $e_0 \in \text{Exp}_{\text{Conf}}$ then one of the following holds with regards to the reduction of $\text{letrec } \{\}$ in e_0 .*

1. $\text{letrec } \{\}$ in e_0 always gets stuck.
2. $\text{letrec } \{\}$ in $e_0 \xrightarrow{\text{R}}^* \text{letrec } H''$ in x if and only if $\text{letrec } \{\}$ in $e_0^\circ \xrightarrow{\text{R}}^* \text{letrec } H'''$ in x with $\text{result}(H'', x) = \text{result}(H''', x)$ and $\text{letrec } \{\}$ in e_0 gets stuck if and only if $\text{letrec } \{\}$ in e_0° 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 $e_1 \xrightarrow{\text{R}}^* \text{letrec } H$ in x if and only if $\text{letrec } \{\}$ in $e_1^\circ \xrightarrow{\text{R}}^* \text{letrec } H'$ 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 $e_1 \xrightarrow{\text{R}}^* \text{letrec } H' \uplus \{x \mapsto \mathbf{weak } y\}$ in x so consequently $\text{letrec } \{\}$ in $\text{ifdead } e_1 (e e_2) e$ always gets stuck or else reduces to $\text{letrec } H' \uplus \{x \mapsto \mathbf{weak } y\}$ in $(e e_2)$. Since $(e e_2)$ is unevaluated $\text{letrec } H' \uplus \{x \mapsto \mathbf{weak } y\}$ in $(e e_2) \xrightarrow{\text{R}}^* \text{letrec } H''$ in e'' iff $\text{letrec } \{\}$ in $(e e_2) \xrightarrow{\text{R}}^* \text{letrec } H'''$ in e''' 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 $e_1 \xrightarrow{\text{R}}^* \text{letrec } H' \uplus \{x \mapsto \mathbf{weak } y\}$ in x if and only if $\text{letrec } \{\}$ in $e_2 \xrightarrow{\text{R}}^* \text{letrec } H'$ in x by Lemma 4.5. From here it is clear that we have $\text{letrec } \{\}$ in $\text{ifdead } e_1 (e e_2) e \xrightarrow{\text{R}}^* \text{letrec } H''$ in x if and only if $\text{letrec } \{\}$ in $e_2 \xrightarrow{\text{R}}^* \text{letrec } H'''$ 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 3.7. From Lemma 4.6 we know that `letrec {}` in e always gets stuck, or it yields the same result as the evaluation of `letrec {}` in e° . Since e° contains no ifdead-expressions, the evaluation of `letrec {}` in e° is deterministic, therefore the evaluation of `letrec {}` in e is deterministic. \square

Furthermore, since no well-typed program can evaluate to a stuck term [HK05], this result shows that any well typed program expression e has the same semantics as e° .

5 A Natural 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 try to 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. We assume we are working within a typed setting, so that we do not have to worry about stuck programs. For ease of reference the typing rules for λ_{weak} are given in Figure 6 and the syntax of types is as follows:

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

5.1 Local Well-behavedness

We say a program `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 5.1 (Loc_Γ). $(e_1, e_2, e_3) \in \text{Loc}_\Gamma$ iff for H such that $\vdash H : \Gamma$

If `letrec H` in $e_1 \xrightarrow{R,*} \text{letrec } H' \uplus \{x \mapsto \mathbf{weak } y, y \mapsto hv\}$ in x then we have

$$\begin{aligned} & \text{letrec } H' \uplus \{x \mapsto \mathbf{weak } y, y \mapsto hv\} \text{ in } e_3 \text{ } y \xrightarrow{R,*} \text{letrec } H'' \text{ in } z \\ \text{iff } & \text{letrec } H' \uplus \{x \mapsto \mathbf{weak } y, y \mapsto hv\} \text{ in } e_2 \xrightarrow{R,*} \text{letrec } H''' \text{ in } z \end{aligned}$$

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

Definition 5.2. A closed program `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, $\Gamma \vdash \mathbf{ifdead } e_1 \ e_2 \ e_3 : \tau'$, 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 5.3. *If a program P is gc-oblivious and well-typed, then it is locally well-behaved.*

Proof. 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 5.4. *If a program P is locally well-behaved then it is well-behaved.*

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

EXAMPLE 5.5 (MEMOIZING FUNCTIONS). Assume we have a type $\mathbf{memofun}(\tau_1, \tau_2)$ of memoized functions from τ_1 to τ_2 with the following functions:

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

$\mathit{Lookupmemo}$ and $\mathit{Addmemo}$ 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 ref , $\mathit{Lookupmemo}$ and $\mathit{Addmemo}$ have the appropriate types. This is not possible because we rely on the dynamic semantics of $\mathit{Addmemo}$ to make the argument, not merely its type. So we assume that $\mathit{Addmemo}$ is inlined. Then we need $\mathit{letrec} H \text{ in } ((\lambda x. \langle x, f \rangle) y) \xrightarrow{R^*} \mathit{letrec} H' \text{ in } z$ iff $\mathit{letrec} H \text{ in } \mathit{Addmemo}(f, o) \xrightarrow{R^*} \mathit{letrec} H'' \text{ in } z$ with $\mathit{result}(H', z) = \mathit{result}(H'', z)$. If ref is dead then $\mathit{letrec} H \text{ in } \mathit{Addmemo}(f, o)$ re-adds the corresponding dead entry, if ref is still alive then $\mathit{letrec} H \text{ in } ((\lambda x. \langle x, f \rangle) y)$ still has the corresponding entry and returns the same pair that $\mathit{letrec} H \text{ in } \mathit{Addmemo}(f, o)$ does. So this example is locally well-behaved, so it is well-behaved. \square

6 Collecting Reachable Weakly-Referenced Garbage

As was proven in [MFH95], 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 safely collected. For example the program $\mathit{letrec} \{x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto \langle x_2, x_2 \rangle, x_4 \mapsto \langle x_1, x_3 \rangle\}$ in $\pi_1 x_4$ is equivalent to the program $\mathit{letrec} \{x_1 \mapsto 1, x_3 \mapsto 0, x_4 \mapsto \langle x_1, x_3 \rangle\}$ 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.

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

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.

Theorem 6.1 (Preservation).

If there exists a typing $\Gamma \vdash e : \tau$ and for some $\vdash H : \Gamma$, we have $\mathit{letrec} H \text{ in } e \xrightarrow{R} \mathit{letrec} H' \text{ in } e'$ 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

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

Figure 6: Typing rules for λ_{weak}

We then use type inference to generate a most general typing for a given program, according to the typing rules in Figure 6. 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 which constrain the type of a reference).

Definition 6.2. 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 . □

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

Lemma 6.3. *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. □

The addition of the type τ **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 are a proper subset of the behaviors of the original program. The following proof is due to Morrisett¹[Mor05].

Theorem 6.4 (Inference GC).

Let $\Gamma_1 = \{x_1 : t_1, \dots, x_n : t_n\}$ and $H_1 = \{x_1 \mapsto h_1, \dots, x_n \mapsto h_n\}$ and $H'_1 = \{x_1 \mapsto 0, \dots, x_n \mapsto 0\}$. If

1. $\Gamma_1 \uplus \Gamma_2 \vdash e : \tau$ ($\tau \notin Tvar$), and
2. $\Gamma_1 \vdash H_2 : \Gamma_2$, and

¹As pointed out by Morrisett, the same proof technique can be used to establish a similar result, Theorem 5.3 in [MFH95], whose original proof was a far more complicated argument using logical relations.

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 (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

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

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 6.1. By Lemma 6.3, 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:

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

If $x \notin FV(e)$ then type-inference would allow us to collect x_2 in this case, 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 \mathbf{d}' . A weak reference that has been replaced with \mathbf{d}' should be treated as alive for the purpose of **ifdead** reduction. A weak reference must only be tombstoned as \mathbf{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**

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

and we change the **ifdead** reduction rule to be

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

We also use an additional typing rule, which assigns to \mathbf{d}' the type $t \mathbf{weak}$ for a type variable t . We require that \mathbf{d}' be typed by a variable in order for Theorem 6.1 to still be true with the new reduction rule for **ifdead**.

We can then 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 \mathbf{d}' without affecting the result of the program.

Theorem 6.5 (Inference Weak GC).

Let $\Gamma_1 = \{x_1 : t_1, \dots, x_n : t_n\}$,
 $H_1^s = \{x_1 \mapsto h_1, \dots, x_n \mapsto h_n\}$,
 $H_1^w = \{y_1 \mapsto \mathbf{weak} x_{i_1}, \dots, y_m \mapsto \mathbf{weak} x_{i_m}\}$,
 $H_1^{ts} = \{x_1 \mapsto 0, \dots, x_n \mapsto 0\}$,
 $H_1^{tw} = \{y_1 \mapsto \mathbf{d}', \dots, y_m \mapsto \mathbf{d}'\}$,
 $H_1 = H_1^s \uplus H_1^w$, and
 $H_1' = H_1^{ts} \uplus H_1^{tw}$.

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. $\mathbf{letrec} H_1 \uplus H_2$ in e is well behaved

then $\mathbf{letrec} H_1 \uplus H_2$ in $e \equiv \mathbf{letrec} H_1' \uplus H_2$ in e .

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

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

If y_i had been tombstoned to \mathbf{d}' we would have

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

Since x_k is assigned a type variable in Γ_1 , by Theorem 6.1 we can still assign it a type variable when typing $\mathbf{letrec} H \uplus \{y_i \mapsto \mathbf{weak} x_k\}$ in $e_2 x_k$, so we can replace the binding of x_k with 0 without affecting the reduction of the program. Therefore $\mathbf{letrec} H_1 \uplus H_2$ in $e \equiv \mathbf{letrec} H_1' \uplus H_2$ in e . \square

7 Key/Value Weak References

In this section we formalize the key/value weak references found in Haskell. To simplify things we do not consider finalizers. A key/value weak reference is a special type of weak reference which contains both a key and a value. During pointer garbage collection, the tracer does not trace the value of a weak pointer unless the key is otherwise reachable. In the GHC documentation [GHC], the semantics is specified as follows:

The behaviour is simply this:

- If a weak pointer (object) refers to an unreachable key, it may be finalised.
- Finalisation means (a) arrange that subsequent calls to `deRefWeak` return `Nothing`; and (b) run the finaliser.

This behaviour depends on what it means for a key to be reachable. 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 `weak e` with `KVweak(e1, e2)` 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:

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

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 $S \cap \text{Dom}(f)$. 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.

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

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

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

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

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

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' .

$$C'_{H'}(H) = H \cup \{z \mapsto H'(z) \mid \exists x, y \in \text{Dom}(H). H(x) = \text{KVweak}(y, z)\}$$

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.

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

Say $R' = \{\text{alloc}, \pi_1, \pi_2, \text{app}, \text{ifdead}, \text{garb}'\}$ and $R'' = \{\text{alloc}, \pi_1, \pi_2, \text{app}, \text{ifdead}, \text{garb}''\}$. Then we have the following.

Theorem 7.1. *For all expressions, e , in the key/value weak calculus we have*

$$\text{letrec } H \text{ in } e \xrightarrow{R'} \text{letrec } H' \text{ in } x \text{ iff } \text{letrec } H \text{ in } e \xrightarrow{R''} \text{letrec } H'' \text{ in } x$$

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

Proof. 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, these differences do not effect the result of the program. \square

8 Conclusion

Related Work

Most of the work related to weak references is in actual implementations of programming languages. Almost every programming language that has garbage collection has some facility for weak references. Aside from the paper on weak references in Haskell [JME00], which contains no formal semantics, the other work on weak references seems to be only of the language reference manual variety.

Summary and Future Work

In this paper we address the correct 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 interesting, as it is much simpler than the previous proof given in [HK05]. 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 his head in order to see what the final outcome of his program will be.

We also extend type-inference based GC to allow collection of additional weak references without incurring computational overhead to recompute data stored in them. We have also shown the flexibility of the semantic framework by extending it to the case of the key/value weak references found in Haskell.

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 like reference mutation and finalization.

References

- [GHC] The Hugs/GHC Team. *Hugs/GHC Extension Libraries: Weak.*
<http://www.dcs.gla.ac.uk/fp/software/ghc/lib/hg-libs-15.html>.

- [HK05] Joseph J. Hallett and Assaf J. Kfoury. A formal semantics for weak references. Technical Report Hal+Kfo:BUCS-TR-2005-X, Department of Computer Science, Boston University, May 2005.
- [JME00] Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. 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*, pages 37–58, London, UK, 2000. Springer-Verlag.
- [MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 66–77, New York, NY, USA, 1995. ACM Press.
- [Mor05] Greg Morrisett, 2005. private communication.
- [SML] The SMLofNJ Team. *SML of NJ Structure Documentation: The WEAK signature*. <http://www.smlnj.org/doc/SMLofNJ/pages/weak.html>.