

# Formal Semantics of Weak References

Kevin Donnelly J. J. Hallett Assaf Kfoury

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

## 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  $\lambda_{\text{weak}}$  that includes weak references and is derived from Morrisett, Felleisen, and Harper's  $\lambda_{\text{gc}}$ . 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** F.3.2 [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 references that do not prevent the object they point to from being garbage collected. During garbage collection, if an object is only reachable through weak references then it may be collected and the weak references to it replaced with a special tag, sometimes called a tombstone. 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.”[12]. 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 [12] 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISMM'06 June 10–11, 2006, Ottawa, Ontario, Canada.  
Copyright © 2006 ACM 1-59593-221-6/06/0006...\$5.00.

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  $\lambda_{gc}$  [10]. 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 is 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 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,  $\lambda_{weak}$ , using a style of definition proposed in [10]. Though quite simple in its final formulation, the design of  $\lambda_{weak}$  was not obvious or necessarily dictated by the use of weak references in practice, if only because there is quite a bit of divergence between the ways in which different programming languages handle them. We strived to define a minimal calculus that can be augmented, or adjusted minimally, to model weak references in more than one language.

In Section 2, we define the syntax and semantics of  $\lambda_{weak}$  and prove several preliminary results. A fundamental aspect of  $\lambda_{weak}$  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.

Section 3 substantiates our claim that  $\lambda_{weak}$  is flexible enough for adaptation to other languages that support weak references. In this section we show how  $\lambda_{weak}$  can be adapted to model the weak references found in Haskell.

In section 4 we describe a more general semantics which allows weak references to be tombstoned at any time, even if the object they point to is still strongly reachable. A programmer making use of this semantics cannot rely on a weak reference being alive at any particular time. This means that compiler optimizations, like common subexpression elimination (CSE), which may cause weak references to go dead sooner than expected, will have behavior that is allowed by this semantics.

In Section 5 we set up a type system for  $\lambda_{weak}$  which, in addition to enforcing the standard invariants, i.e., catching programs that “go wrong” (Subsection 5.1), can be used for a more efficient management of memory (Subsection 5.2). We extend the latter 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 6 we study the conditions under which  $\lambda_{weak}$  programs are “well-behaved”, i.e., under which garbage collection does not affect the result of evaluation. It is undecidable whether an arbitrary  $\lambda_{weak}$  program is well-behaved.

In Subsection 6.1 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 6.2 we define a general, but undecidable, criterion for the well-

behavedness of programs which can be used as a guideline for writing programs satisfying the property.

In Section 7 we discuss related work and in Section 8, we propose several directions for future research and conclude. Missing proofs and additional materials can be found in the two companion reports, [4, 5].

The appendix contains the proofs and auxiliary lemmas of theorems in Sections 4 and 5.

## 2. Modeling Weak References: $\lambda_{weak}$

A formal model called  $\lambda_{weak}$ , which extends  $\lambda_{gc}$  with the means to introduce and conditionally dereference weak references, is given in [5] and further investigated in [4].  $\lambda_{weak}$  formalizes the semantics for weak references in SML/NJ [12] by giving a semantics to weak references in which garbage collection is non-deterministically applied. Therefore a programmer using  $\lambda_{weak}$  to reason about his program cannot rely on the timing of garbage collection. This is a natural way to reason about weak references as the programmer (usually) is not aware of when garbage collection occurs. In Section 4 we briefly explore another semantics which, in addition to non-deterministically applying garbage collection, non-deterministically tombstones weak references. With this semantics a programmer cannot rely on the liveness of a weak reference, but in return we can model the semantics of programs that are optimized by compilers.

### Syntax of $\lambda_{weak}$

The syntax of  $\lambda_{weak}$  (given in Figure 1) is that of a standard programming language based on the  $\lambda$ -calculus along with additional primitives for introducing weak references and doing conditional weak dereferencing. A  $\lambda_{weak}$  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  $(ifdead e_1 e_2 e_3)$ . In some examples we make use of let  $x = e_1$  in  $e_2$  as syntactic sugar for  $(\lambda x. e_2) e_1$ .

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. When this happens we say the weak pointer has been *tombstoned*.

A  $\lambda_{weak}$  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  $\text{Dom}(H)$  and  $\text{Ran}(H)$  to be the domain and range of  $H$  and we define

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

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 heap  $H$  and a program letrec  $H$  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  $\alpha$ -conversion and programs are identified up to renaming of variables bound in the heap, e.g., letrec  $H \uplus \{x \mapsto hv\}$  in  $x = \text{letrec } H \uplus \{y \mapsto hv\{x := y\}\}$  in  $y$  assuming  $x \notin FV(H)$  and  $y \notin FV(H)$ .

---

<b>Programs:</b>			
(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)	$e \in \text{Exp}$	$::=$	$x \mid i \mid \langle e_1, e_2 \rangle \mid \pi_1 e \mid \pi_2 e \mid \lambda x. e \mid e_1 e_2 \mid$ $\text{weak } e \mid \text{ifdead } e_1 e_2 e_3$
(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)	$E \in \text{Ctxt}$	$::=$	$[] \mid \langle E, e \rangle \mid \langle x, E \rangle \mid \pi_i E \mid E e \mid x E \mid \text{weak } E \mid \text{ifdead } E e_1 e_2$
(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
( $\pi_i$ )	$\text{letrec } H \text{ in } E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H \text{ in } E[x_i]$
	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  $\lambda_{\text{weak}}$

**Semantics of  $\lambda_{\text{weak}}$**

The reduction semantics of  $\lambda_{\text{weak}}$  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 ( $\pi_i$ ) 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 dereference 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  $\text{letrec } H \text{ in } e_2$ . Unlike Standard ML which separates these two actions by dereferencing to an option datatype and using case analysis to branch on the result,  $\lambda_{\text{weak}}$  combines these so we do not have to introduce datatypes. This combination also allows us to syntactically identify programs whose evaluation does not depend on garbage collection, which we explore in Section 6.

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:

$$\text{(garb)} \quad \text{letrec } H \text{ in } e \xrightarrow{\text{garb}} \text{letrec } H' \text{ in } e$$

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 garbage<sup>1</sup>. 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, the garbage collector may be implemented 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 extreme 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 rule we are sure that the programmer cannot rely on some location having been collected, so it is safe to perform many compiler optimizations which make object identity statically unknown.

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  $\lambda_{\text{weak}}$ .

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

<sup>1</sup> An 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).

EXAMPLE 2.1.

$$\begin{array}{l}
P_1 = \text{letrec } \{ \} \text{ in } (\lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y)) \langle 5, 6 \rangle \\
\text{alloc}^* \rightarrow \text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle \} \text{ in } a e \\
\text{app} \rightarrow \text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle \} \text{ in ifdead } (\text{weak } e) 0 (\lambda y. \pi_1 y) \\
\text{alloc} \rightarrow \text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle, f \mapsto \text{weak } e \} \text{ in ifdead } f 0 (\lambda y. \pi_1 y) \\
\text{then} \\
\text{ifdead} \rightarrow \text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle, f \mapsto \text{weak } e \} \text{ in } (\lambda y. \pi_1 y) e \longrightarrow \dots \xrightarrow{\text{garb}} \text{letrec } \{ b \mapsto 5 \} \text{ in } b \\
\text{or} \\
\text{garb} \rightarrow \text{letrec } \{ f \mapsto d \} \text{ in ifdead } f 0 (\lambda y. \pi_1 y) \longrightarrow \dots \xrightarrow{\text{garb}} \text{letrec } \{ g \mapsto 0 \} \text{ in } g
\end{array}$$

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

Figure 2. Example of Non-confluent Reduction

ple:

$$P_2 = \text{letrec } \{ \} \text{ in } (\lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda z. \text{ifdead } (\text{weak } z) \Omega (\lambda y. \pi_1 y))) \langle 5, 6 \rangle$$

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

If a program is completely evaluated without getting stuck, it will have the form:  $\text{letrec } H \text{ in } x$ . In order to talk about the value in the heap that  $x$  represents, we define  $\text{result}(S, H, e)$  as in Figure 3. We use this function to recursively replace any heap locations in  $e$  with the heap value it is mapped to in  $H$ . Note that cycles in the heap may cause problems. For example, the value in the heap  $\{x \mapsto \langle x, x \rangle\}$  that  $x$  represents is undefined. However, we can detect cycles by keeping track of the heap locations that have been visited as we traverse the heap. If a cycle is detected, we set  $\text{result}(S, H, e) = \bullet$ . We use  $\text{result}(H, e)$  as short-hand notation for  $\text{result}(\emptyset, H, e)$ .

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  $\lambda_{\text{weak}}$  program is stuck if it is of one of the following forms:

$$\begin{array}{l}
\text{letrec } H \text{ in } E[\pi_i x] \\
(x \notin \text{Dom}(H) \text{ or } H(x) \neq \langle x_1, x_2 \rangle)
\end{array}$$

$$\begin{array}{l}
\text{letrec } H \text{ in } E[x y] \\
(x \notin \text{Dom}(H) \text{ or } H(x) \neq \lambda z. e)
\end{array}$$

$$\begin{array}{l}
\text{letrec } H \text{ in } E[\text{ifdead } x e_1 e_2] \\
(x \notin \text{Dom}(H) \text{ or } (H(x) \neq \text{weak } w \text{ and } H(x) \neq d)) \quad \square
\end{array}$$

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

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

$\square$

Because the untyped lambda-calculus is a subset of  $\lambda_{\text{weak}}$ ,  $\text{eval-set}(P, G)$  is an undecidable set in general. 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 [10]. However, if all evaluations of  $P$  and  $P'$  return an int answer *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  $\lambda_{\text{weak}}$  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  $P = \text{letrec } H \uplus \{x \mapsto hv\}$  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  $\beta$ -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  $\beta$ -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 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. GHC Haskell Key/Value Weak References

Our formal setup is flexible enough to handle weak references in other popular programming languages. In this section we formalize the key/value weak references found in GHC Haskell (which are similar to ephemerons [6]) by extending the syntax and replacing the garbage collection rule of  $\lambda_{\text{weak}}$ . We call the garbage collection rule derived from the GHC documentation (garb’).

We write  $\text{result}(H, e)$  for  $\text{result}(\emptyset, H, e)$ .

$$\text{result}(S, H, e) = \begin{cases} x & \text{if } e = x \text{ and } x \notin \text{Dom}(H) \\ \text{result}(S \cup \{x\}, H, H(x)) & \text{if } e = x \text{ and } x \in \text{Dom}(H) \text{ and } x \notin S \\ i & \text{if } e = i \\ d & \text{if } e = d \\ \langle \text{result}(S, H, e_1), \text{result}(S, H, e_2) \rangle & \text{if } e = \langle e_1, e_2 \rangle \text{ and } \text{result}(S, H, e_i) \neq \bullet \text{ for } 1 \leq i \leq 2 \\ \pi_i \text{ result}(S, H, e') & \text{if } e = \pi_i e' \text{ and } \text{result}(S, H, e_i) \neq \bullet \text{ for } 1 \leq i \leq 2 \\ \lambda x. \text{result}(S, H, e') & \text{if } e = \lambda x. e' \text{ and } x \notin \text{Dom}(H) \text{ and } \text{result}(S, H, e') \neq \bullet \\ \text{result}(S, H, e_1) \text{ result}(S, H, e_2) & \text{if } e = e_1 e_2 \text{ and } \text{result}(S, H, e_i) \neq \bullet \text{ for } 1 \leq i \leq 2 \\ \text{weak result}(S, H, e') & \text{if } e = \text{weak } e' \text{ and } \text{result}(S, H, e') \neq \bullet \\ \text{ifdead result}(S, H, e_1) \text{ result}(S, H, e_2) \text{ result}(S, H, e_3) & \text{if } e = \text{ifdead } e_1 e_2 e_3 \text{ and } \text{result}(S, H, e_i) \neq \bullet \text{ for } 1 \leq i \leq 3 \\ \bullet & \text{otherwise} \end{cases}$$

**Figure 3.** Definition of  $\text{result}(S, H, e)$

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 GHC). 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 finalizer of an object whose key is reachable.

Notice that a pointer to the key from its associated value or finalizer 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.

To formalize key/value weak references 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:

$$\begin{aligned} C_{H'}(H) = & H \cup \{z \mapsto H'(z), x \mapsto \text{KVweak}(y, z) \mid \\ & y \in \text{Dom}(H) \wedge H'(x) = \text{KVweak}(y, z)\} \\ & \cup \{x \mapsto H'(x) \mid \exists e. e \in \text{Ran}(H) \wedge \\ & x \in \text{FV}(e) \wedge \forall yz. e \neq \text{KVweak}(y, z)\} \end{aligned}$$

We define the reachable part of the heap,

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

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 GHC 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

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

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

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

According to the semantics of GHC, a key/value weak pointer is reachable if its key is reachable, even if the weak pointer object itself is unreachable. We can use (garb') to illustrate this. For example, in the program:

$$\text{letrec } \{y \mapsto \text{KVweak}(x, z), z \mapsto (\lambda x.e)\} \text{ in } x$$

the location  $z$  may not be garbage collected because it is reachable according to the semantics.

## 4. Semantics to Encompass Compiler Optimizations

As briefly discussed, compiler optimizations can affect the reachability of heap values. Consider the program:

$$\text{letrec } \{a \mapsto \lambda x. \lambda y. x, b \mapsto 6, x \mapsto \langle a, b \rangle, y \mapsto \text{weak } x, f \mapsto \lambda x. x\} \text{ in } f(\pi_1 x) (\text{ifdead } y \ 2 (\lambda z. \pi_1 z)) (\pi_1 x).$$

Most optimizing compilers would eliminate the common subexpression  $\pi_1 x$ , resulting in the seemingly equivalent program:

$$\text{letrec } \{a \mapsto \lambda x. \lambda y. x, b \mapsto 6, x \mapsto \langle a, b \rangle, y \mapsto \text{weak } x, f \mapsto \lambda x. x\} \text{ in let } c = \pi_1 x \text{ in } f \ c (\text{ifdead } y \ 2 (\lambda z. \pi_1 z)) \ c$$

In the optimized program,  $x$  becomes unreachable after the let binding is evaluated. If garbage collection occurs at this point, the program will return the value 2, a result that is impossible in the program before optimization.

In the presence of such compiler optimizations the (garb) rule of Section 2 does not correctly model reachability-based garbage collection and this can affect the semantics of a program. One approach to remedying this situation is to write a compiler that produces code that obeys the rules of  $\lambda_{\text{weak}}$ . In the example above,

the compiler would not be able to perform CSE on the first and third arguments. Another approach is to generalize the definition of (garb) so that the weak reference to  $x$  in the first example above can be tombstoned. We can realize the second approach by adding another rule to arbitrarily tombstone weak references.

$$\text{(garb'')} \quad \text{letrec } H \uplus H' \text{ in } e \xrightarrow{\text{garb''}} \\ \text{letrec } H \uplus \{x_1 \mapsto \mathbf{d}, \dots, x_n \mapsto \mathbf{d}\} \text{ in } e \\ \text{provided } H' = \{x_1 \mapsto \mathbf{weak } y_1, \dots, x_n \mapsto \mathbf{weak } y_2\}$$

This rule allows weak references to be tombstoned at any time. Therefore not only can common subexpressions be collected earlier than expected, but any weakly referenced value can. With this rule the programmer can no longer rely on syntactic reachability to determine when a weakly referenced value will be collected. However, this is an inherent effect of compiler optimizations that may alter reachability. In Section 6 we discuss methods for determining that garbage collection cannot change the result of a program. The methods that we give do not analyze the reachability of weakly referenced values, so they are suitable for reasoning about behavior in the presence of (garb'').

## 5. Types for Garbage Collection

As in [10] 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 unnecessary overhead of recomputing unused data cached in the weak references.

### 5.1 Monomorphic Type System

In this section we introduce a standard monomorphic type system for  $\lambda_{\text{weak}}$ . There are no surprises here. While we could formulate an explicitly typed language allowing for tag-free garbage collection [1, 14], this was already done in [10] and there are no additional complications arising from weak references. Therefore we will formulate an implicit type assignment system for the language  $\lambda_{\text{weak}}$  already defined.

The syntax of types is as follows.

#### Types:

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

The typing rules are shown in Figure 4 and are (almost) completely standard. The one non-standard addition is that we assign to  $\mathbf{d}$  the type  $\tau \text{ weak}$  for any  $\tau$ . 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 [15], whose proofs are standard.

**Lemma 5.1** (Progress). *For every  $\lambda_{\text{weak}}$  program  $P$ , if  $\vdash P : \tau$  then either  $P$  is an answer or there exists  $P'$  such that  $P \xrightarrow{\text{R}} P'$ .*  $\square$

**Lemma 5.2** (Preservation). *For every  $\lambda_{\text{weak}}$  program  $P$ , if  $\vdash P : \tau$  and  $P \xrightarrow{\text{R}} P'$  then  $\vdash P' : \tau$ .*  $\square$

**Theorem 5.3** (Type Soundness). *For every  $\lambda_{\text{weak}}$  program  $P$ , if  $\vdash P : \tau$  then either  $P$  is an answer or else there is some  $P'$  such that  $P \xrightarrow{\text{R}} P'$  and  $\vdash P' : \tau$ .*  $\square$

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

### 5.2 Collecting Reachable Garbage

As was pointed out in [2], and proven in [10], 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  $\lambda_{\text{weak}}$ , to be an implicitly typed language with type variables.

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

The presence of type variables allows us to derive a *most general type* for each well-typed program, i.e., a type such that every other type 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 5.4** (Preservation). *If there exists a typing  $\Gamma \vdash e : \tau$  and for some  $\vdash H : \Gamma$ , we have  $\text{letrec } H \text{ in } e \xrightarrow{\text{R}} \text{letrec } H' \text{ in } e'$  then there exists  $\Gamma'$  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 type 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 5.5.** 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, if a reference is assigned a type variable then it cannot appear in an active position.

**Lemma 5.6.** *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** <sub>$i$</sub> ), (**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 weak references has a small effect on the correctness of inference GC. Because, in general, garbage collection can affect the result of a program, we need to assume that the program we are running garbage collection on is well-behaved in order to prove that inference GC preserves the semantics. Proof of the following theorem can be found in the Appendix A. The proof technique is suggested at the end of [10].

**Theorem 5.7** (Inference GC). *Let*

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

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

Figure 4. Typing rules for  $\lambda_{\text{weak}}$

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 (i.e. the timing of garbage collection cannot affect the final result)

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

So type-inference based GC works in this language. We can actually get a bit more traction out of inference GC in the presence of weak references. Often weak pointers are used to cache data that was computationally expensive to produce, so killing a weak pointer may cause expensive recomputation. Inference GC allows for additional garbage to be collected and therefore additional weak references to be tombstoned. Because of the type inference, we know that values pointed to by these additional weak references will never be used, so recomputing (by taking the dead branch of an ifdead-expression) the value is pointless. Consider the following program:

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

If  $x \notin FV(e)$  then type-inference would allow us to collect  $x_2$  (because then we can assign  $x_5 : t$  weak and  $x_2 : t$ ), which would cause  $x_5$  to be tombstoned. This means 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.

We can avoid this useless recomputation by adding a new distinct tombstone marker  $d'$ . A weak reference that has been replaced with  $d'$  will 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

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

and we change the ifdead reduction rule to be

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

We also use an additional typing rule, which assigns to  $d'$  the type  $t$  weak for a type variable  $t$ . We only need to allow  $d'$

to be typed by  $t$  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 5.4 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. The proof can be found in Appendix A.

**Theorem 5.8** (Inference Weak GC). *Let*

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

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

## 6. 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. For these programs, which we believe are ubiquitous, it is desirable to be able to check whether garbage collection can influence the result. Much as a type system is used to prove programs “don’t go wrong” this section describes techniques to prove garbage collection cannot alter the result of evaluation.

We refer to programs which always evaluate to the same result as *well-behaved* (note this is weaker than the usual notion of confluence). The property is formally given by Definition 2.7 and its undecidability shown in Proposition 2.8. 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  $\text{letrec } H \text{ in } e'$ . Assuming there are no occurrences of

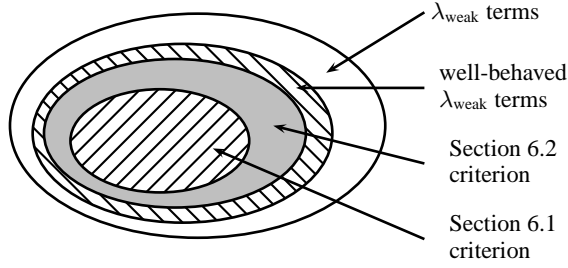


Figure 5. Well-behavedness criteria

weak  $e''$  in  $e$  or  $e'$  for any  $e''$ , all reductions of this program must end with the same result.

In this section we give two criteria for determining that a program is well behaved. First we give a decidable syntactic criterion, then a less restrictive but undecidable semantic criterion. Figure 5 shows the relationships of these criteria. It will be interesting (left for future research) to adjust the syntactic criterion of Section 6.1, or define a new syntactic criterion, that decreases the gap to the semantic criterion of Section 6.2.

### 6.1 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  $\text{Exp}^*$  is defined in Figure 6. 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 “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 6.1** (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 6.2** (GC-Oblivious Programs). A program  $\text{letrec } \{ \}$  in  $e$  is *gc-oblivious* iff  $e \in \text{Exp}^*$ .  $\square$

**Proposition 6.3.** *Membership in  $\text{Exp}^*$  is recognizable in linear time.*

*Proof.* The rules for  $\text{ExpPair}$  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 6.4.** *If  $P$  is gc-oblivious and well-typed then it is well-behaved.*  $\square$

A complete proof of this theorem and all auxiliary lemmas can be found in Appendix B. Below we give a sketch of the pertinent definitions and lemmas of the proof.

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 [10].

### Generalization of GC-Obliviousness

We will generalize the notion of gc-obliviousness for well-typed terms by defining a set  $\text{ExpConf}$ , as in Figure 7, which contains all well-typed gc-oblivious terms.

**Definition 6.5** (Relation  $\text{Conf}_\tau$ ). For  $e_1, e_2$ , such that  $\Gamma \vdash e_1 : \tau_1 \rightarrow \dots \rightarrow \tau_n$  weak and  $\Gamma \vdash e_2 : \tau_1 \rightarrow \dots \rightarrow \tau_n$

1.  $(e_1, e_2) \in \text{Conf}_{\tau \text{ weak}}$  if  $\Gamma \vdash e_1 : \tau$  weak and for all  $H$  such that  $\vdash H : \Gamma$

$$\text{letrec } H \text{ in } e_1 \xrightarrow{R}^* \text{letrec } H' \uplus \{x \mapsto \text{weak } y\} \text{ in } x \\ \text{iff } \text{letrec } H \text{ in } e_2 \xrightarrow{R}^* \text{letrec } H' \text{ in } y.$$

2.  $(e_1, e_2) \in \text{Conf}_{\tau_1 \rightarrow \tau_2}$  if  $\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2$  and for all  $e \in \text{Exp}^*$  with  $\Gamma \vdash e : \tau_1$  we have  $(e_1 e, e_2 e) \in \text{Conf}_{\tau_2}$ .  $\square$

**Lemma 6.6.**  $(e_1, e_2) \in \text{ExpPair}$  and  $\Gamma \vdash e_1 : \tau$  implies  $(e_1, e_2) \in \text{Conf}_\tau$   $\square$

We define the transformation  $e^\circ$  as follows:

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

**Lemma 6.7.** *Suppose  $e_0 \in \text{ExpConf}$  and  $\Gamma \vdash e_0 : \tau$  then for any  $H$  such that  $\vdash H : \Gamma$ ,  $\text{letrec } H \text{ in } e_0 \equiv \text{letrec } H \text{ in } e_0^\circ$ .*  $\square$

*Proof of Theorem 6.4.* From Lemma 6.7 we know that for closed, well-typed  $e$ ,  $\text{letrec } \{ \}$  in  $e$  yields the same possible results as the evaluation of  $\text{letrec } \{ \}$  in  $e^\circ$ . Since  $e^\circ$  contains no ifdead-expressions, the final result of  $\text{letrec } \{ \}$  in  $e^\circ$  is unique, therefore the final result of  $\text{letrec } \{ \}$  in  $e$  is unique.  $\square$

### 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  $\text{ExpPair}$  relation with  $p \in \{1, 2\}^*$  to obtain a larger set of gc-oblivious programs that includes pairs and projections. 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 8. We would then change the definition of  $\text{Exp}^*$  to use  $\text{ExpPair}(\varepsilon)$  in place of  $\text{ExpPair}$ , where  $\varepsilon$  is the empty sequence.

All of the nice properties of the  $\text{ExpPair}$  relation still hold for  $\text{ExpPair}(p)$ . In particular, the proofs of Proposition 6.3 and Theorem 6.4 easily generalize. For example, notice that the definition of the  $\text{Conf}$  relation naturally generalizes when we expand the set of gc-oblivious programs to handle pairing and projection using the parameterized relation  $\text{ExpPair}(p)$ . In this case we parameterize  $\text{Conf}$  as  $\text{Conf}(p)$ .

### 6.2 A General Semantic Criterion for Well-Behavedness

The syntactic restriction of gc-obliviousness in Section 6.1 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

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

Figure 6. Definition of  $\text{Exp}^*$

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

Figure 7. Definition of  $\text{ExpConf}$

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

Figure 8. Extended  $\text{ExpPair}(p)$  Definition

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

**Definition 6.8** ( $\text{Loc}_\Gamma$ ).  $(e_1, e_2, e_3) \in \text{Loc}_\Gamma$  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

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

iff

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

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

**Definition 6.9.** A closed program  $\text{letrec } H \text{ in } e$  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  $(e_1, e_2, e_3) \in \text{Loc}_\Gamma$ .  $\square$

The notion of local well-behavedness is an undecidable property because well-typed programs can contain cyclic heap bindings<sup>2</sup>. For example consider the following well-typed program

$$\text{letrec } \{f \mapsto \lambda x. \lambda y. f x (x y)\} \text{ in } f (\lambda x. x) 3.$$

However this is still a natural criterion to use when programming with weak references.

**Theorem 6.10.** *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 6.11.** *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 6.12** (Memoizing Functions). Assume we have a type  $\text{memofun}(\tau_1, \tau_2)$  of memoized functions from  $\tau_1$  to  $\tau_2$  with the

<sup>2</sup>Note however, that well-typed programs with an empty heap can never reduce to a well-typed program with a cyclic heap binding because the  $(\text{alloc})$  rule guarantees that the left-hand side of a heap binding is fresh.

following functions:

```

Lookupmemo : (memofun( $\tau_1, \tau_2$ ) *  $\tau_1$ )  $\rightarrow$   $\tau_2$  weak option
Addmemo    : (memofun( $\tau_1, \tau_2$ ) *  $\tau_1$ )  $\rightarrow$ 
              ( $\tau_2$  * memofun( $\tau_1, \tau_2$ ))

```

Lookupmemo and Addmemo do not need to use ifdead so they are locally well-behaved. We 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 function should fit our definition of locally well-behaved. We need to prove that each branch of the ifdead expression will produce the same result in any heap  $H$  such that  $\text{ref}$ ,  $\text{Lookupmemo}$  and  $\text{Addmemo}$  have the appropriate types. This is not possible because we need to make use of the dynamic semantics of  $\text{Addmemo}$  to make the argument, not merely its type. In order to make the argument, we assume that the definition of  $\text{Addmemo}$  is available and we reason as if it were inlined. 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  $\text{ref}$  is dead then

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

re-adds the corresponding dead entry, if  $\text{ref}$  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.  $\square$

## 7. Related Work

Morrisett, Felleisen and Harper's [10] was the first work to fully specify the static and dynamic semantics of a language with garbage collection and was followed up by [11], 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 GHC 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.

## 8. 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 GHC 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 of the preceding result 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. We would also like to extend our calculus to formalize variants of weak references which are garbage collected according to different heuristics, similar to those found in Java. A good approach may be to include several types of weak references in our formalism which are treated differently by the garbage collection rule.

## Acknowledgments

Adam Bakewell, Sebastien Carlier, Chiyang Chen, Likai Lui, Peter Møller Neergaard, Greg Morrisett, Franklyn Turbak, Joe Wells, Hongwei Xi and anonymous reviewers all 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. Ephemerons: 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] PEYTON JONES, S. L., 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., 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.
- [11] MORRISSETT, G., AND HARPER, R. Semantics of memory management for polymorphic languages. In *Higher Order Operational Techniques in Semantics*, A. Gordon and A. Pitts, Eds. Newton Institute, Cambridge University Press, 1997.
- [12] THE SMLofNJ TEAM. *SML of NJ Structure Documentation: The WEAK signature*. <http://www.smlnj.org/doc/SMLofNJ/pages/weak.html>.
- [13] 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.

- [14] TOLMACH, A. P. Tag-free garbage collection using explicit type parameters. In *LISP and Functional Programming* (1994), pp. 1–11.
- [15] WRIGHT, A. K., AND FELLEISEN, M. A syntactic approach to type soundness. *Information and Computation* 115, 1 (1994), 38–94.

## A. Proofs for Inference GC

**Theorem A.1** (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
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 5.4. By Lemma 5.6, 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  $\text{letrec } H_1 \uplus H_2 \text{ in } e \equiv \text{letrec } H'_1 \uplus H_2 \text{ in } e$ .  $\square$

**Theorem A.2** (Inference Weak GC). *Let*

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

*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

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

If  $y_i$  had been tombstoned to  $d'$  we would have

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

Since  $x_k$  is assigned a type variable in  $\Gamma_1$ , by Theorem 5.4 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  $\text{letrec } H_1 \uplus H_2 \text{ in } e \equiv \text{letrec } H'_1 \uplus H_2 \text{ in } e$ .  $\square$

## B. Proofs for Recoving Uniqueness of Program Result

For the following lemma we define equivalence of heaps  $H_1, H_2$ , with respect to expressions  $e_1, \dots, e_n$ ,  $H_1 \equiv_{\{e_1, \dots, e_n\}} H_2$  by:

1.  $\text{FV}(e_1, \dots, e_n) \cap \text{Dom}(H_1) = \text{FV}(e_1, \dots, e_n) \cap \text{Dom}(H_2)$  (call this set  $V$ )
2.  $H_1[V] = H_2[V]$ , and
3.  $(H_1 \setminus V) \equiv_{H_1[V]} (H_2 \setminus V)$

where  $(H \setminus V)$  is defined by  $(H \setminus V)(x) = H(x)$  if  $x \notin V$  and undefined otherwise.

So  $H_1 \equiv_{\{e\}} H_2$  if and only if  $H_1$  is equal to  $H_2$  on all of the variables that are reachable from  $e$ .

**Lemma B.1.** *If  $\Gamma \vdash e_1 : \tau = \tau_1 \rightarrow \dots \rightarrow \tau_n$  weak and  $\Gamma \vdash e_2 : \tau_1 \rightarrow \dots \rightarrow \tau_n$  and for all  $H$  such that  $\vdash H : \Gamma$  there are  $e'_1, e'_2$  and  $H'$  such that*

1.  $\text{letrec } H \text{ in } e_i \xrightarrow{R}^* \text{letrec } H' \text{ in } e'_i$ , and
2. All evaluations of  $\text{letrec } H \text{ in } e_i$  lead to  $\text{letrec } H'' \text{ in } e'_i$  (for some  $H''$ ) with  $H'' \equiv_{\{e_1, e_2\}} H'$
3.  $(e'_1, e'_2) \in \text{Conf}_\tau$

then  $(e_1, e_2) \in \text{Conf}_\tau$

*Proof Sketch.* By induction on  $\tau$ .

The conditions imply that all reductions of  $\text{letrec } H \text{ in } e_i$  are equivalent to an extension of the given reductions

$$\text{letrec } H \text{ in } e_i \xrightarrow{R}^* \text{letrec } H' \text{ in } e'_i.$$

For  $\tau = \tau_1$  weak the result is immediate because all reductions of  $\text{letrec } H \text{ in } e_1$  and  $\text{letrec } H \text{ in } e_2$  are equivalent to reductions of  $\text{letrec } H' \text{ in } e'_1$  and  $\text{letrec } H' \text{ in } e'_2$  and membership in  $\text{Conf}_{\tau_1 \text{ weak}}$  depends only on the final results of reductions.

For  $\tau = \tau' \rightarrow \tau''$ , because  $\square e$  is an evaluation context, the reductions above are also reductions of  $\text{letrec } H \text{ in } e_i \ e$  to  $\text{letrec } H \text{ in } e'_i \ e$ , so we can apply the IH to get  $(e_1 \ e, e_2 \ e) \in \text{Conf}_{\tau''}$ .  $\square$

For using the above lemma, note that if we have evaluations

$$\begin{aligned}\text{letrec } H \text{ in } e_1 &\xrightarrow{R\text{-}\{\text{ifdead}\}}^* \text{letrec } H' \text{ in } e'_1 \\ \text{and letrec } H \text{ in } e_2 &\xrightarrow{R\text{-}\{\text{ifdead}\}}^* \text{letrec } H' \text{ in } e'_2\end{aligned}$$

in which no weakly referenced garbage gets collected then all evaluations of  $\text{letrec } H \text{ in } e_i$  lead to  $\text{letrec } H'' \text{ in } e'_i$  with  $H'' \equiv \{e_1, e_2\} H'$ .

**Lemma B.2.**  $(e_1, e_2) \in \text{ExpPair}$  and  $\Gamma \vdash e_1 : \tau$  implies  $(e_1, e_2) \in \text{Conf}_\tau$

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

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

By inversion on derivation of  $\Gamma \vdash \text{weak } e : \tau, \tau = \tau' \text{ weak}$  for some  $\tau'$ . So we need  $(\text{weak } e, e) \in \text{Conf}_{\tau' \text{ weak}}$ . Since  $\text{weak } \square$  is an evaluation context all evaluations of  $\text{weak } e$  and  $e$  match with the exception of the last step of halting evaluations, which allocate the weak pointer satisfying the definition for  $\text{Conf}_{\tau' \text{ weak}}$ .

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

By inversion on derivation of  $\Gamma \vdash \lambda x.e_1 : \tau, \tau = \tau_1 \rightarrow \tau_2$  for some  $\tau_1, \tau_2$ .

Suppose  $e \in \text{Exp}^*$  and  $\Gamma \vdash e : \tau_1$ . Then we also have  $\Gamma \vdash (\lambda x.e_1) e : \tau_2$  and we need  $((\lambda x.e_1) e, (\lambda x.e_2) e) \in \text{Conf}_{\tau_2}$ . There are reductions

$$\begin{array}{l} \text{letrec } H \text{ in } (\lambda x.e_i) e \xrightarrow{R\text{-}\{\text{ifdead}\}^*} \\ \text{letrec } H' \uplus \{f \mapsto \lambda x.e_i\} \text{ in } f x \end{array}$$

which collect no garbage. The result then steps to  $\text{letrec } H' \uplus \{f \mapsto \lambda x.e_i\} \text{ in } e_i$  and we can garbage collect  $f$ , which cannot be free in  $e_i$  since it was allocated fresh for  $(\lambda x.e_i)$  and it cannot be weakly referenced ( $\lambda x.e_i$ ) was in head position at the time of allocation. By IH  $(e_1, e_2) \in \text{Conf}_{\tau_2}$ , so by Lemma B.1  $((\lambda x.e_1) e, (\lambda x.e_2) e) \in \text{Conf}_{\tau_2}$ .

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

Follows immediately from inversion on typing and the IH.

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

By inversion on  $\Gamma \vdash \text{ifdead } e_1 (e_3 e_2) e_3 : \tau$  we have  $\Gamma \vdash e_1 : \tau_1 \text{ weak}$  and  $\Gamma \vdash e_3 e_2 : \tau$  and  $\Gamma \vdash e_3 : \tau_1 \rightarrow \tau$ .

Suppose  $H$  is a heap with  $\vdash H : \Gamma$ . There are reductions:

$$\begin{array}{l} \text{letrec } H \text{ in ifdead } e_1 (e_3 e_2) e_3 \xrightarrow{R\text{-}\{\text{ifdead}\}^*} \\ \text{letrec } H' \uplus \{x \mapsto \text{weak } y\} \text{ in ifdead } x (e_3 e_2) e_3 \end{array}$$

$$\text{and letrec } H \text{ in ifdead } e_1 (e_4 e_2) e_4 \xrightarrow{R\text{-}\{\text{ifdead}\}^*} \\ \text{letrec } H' \uplus \{x \mapsto \text{weak } y\} \text{ in ifdead } x (e_4 e_2) e_4$$

which collect no garbage, so all reductions are equivalent to extensions of these reductions. All reductions proceed by either collecting some garbage and causing  $x$  to be tombstoned or by taking the live branch of the `ifdead`. By IH on  $(e_1, e_2) \in \text{ExpPair}$ , we have  $(e_1, e_2) \in \text{Conf}_{\tau_1 \text{ weak}}$ . We know  $x$  cannot be free in  $H', e_2, e_3$ , or  $e_4$  since it was allocated fresh. Therefore, by  $(e_1, e_2) \in \text{Conf}_{\tau_1 \text{ weak}}$  all reductions of these are equivalent to reductions of  $\text{letrec } H' \text{ in } e_3 y$  and  $\text{letrec } H' \text{ in } e_4 y$ . By type preservation, we have  $\Gamma' \vdash e_3 y : \tau$  for  $\vdash H' : \Gamma'$ . By IH on  $(e_3, e_4) \in \text{ExpPair}$ , we have  $(e_3, e_4) \in \text{Conf}_{\tau_1 \rightarrow \tau}$  and therefore  $(e_3 y, e_4 y) \in \text{Conf}_{\tau}$ . By lemma B.1  $(\text{ifdead } e_1 (e_3 e_2) e_3, \text{ifdead } e_1 (e_4 e_2) e_4) \in \text{Conf}_{\tau}$ .  $\square$

**Lemma B.3.** *Suppose  $e_0 \in \text{ExpConf}$  and  $\Gamma \vdash e_0 : \tau$  then for any  $H$  such that  $\vdash H : \Gamma$ ,  $\text{letrec } H \text{ in } e_0 \equiv \text{letrec } H \text{ in } e_0^\circ$ .*

*Proof Sketch.* 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.

By inversion on typings we have  $\Gamma \vdash e_1 : \tau_1 \text{ weak}$  and by inversion on  $\text{ExpConf}$  we have  $(e_1, e_2) \in \text{Conf}_{\tau_1 \text{ weak}}$ . Since bindings allocated during evaluation of  $e_1$  cannot effect evaluation of  $e_2$ ,  $(e_1, e_2) \in \text{Conf}_{\tau_1 \text{ weak}}$  tells us  $\text{ifdead } e_1 (e e_2) e \equiv (e e_2)$ . Therefore the programs are equivalent.  $\square$