

# A Formal Semantics for Weak References

J. J. Hallett  
Boston University  
jhallett@cs.bu.edu

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

May 22, 2005

Modified: August 8, 2005

## Abstract

A weak reference is a reference to an object that is not followed by the pointer tracer when garbage collection is called. That is, a weak reference cannot prevent the object it references from being garbage collected. Weak references remain a troublesome programming feature largely because there is not an accepted, precise semantics that describes their behavior (in fact, we are not aware of any formalization of their semantics). The trouble is that weak references allow reachable objects to be garbage collected, therefore allowing garbage collection to influence the result of a program. Despite this difficulty, weak references continue to be used in practice for reasons related to efficient storage management, and are included in many popular programming languages (Standard ML, Haskell, OCaml, and Java).

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  $\lambda_{\text{gc}}$ .  $\lambda_{\text{gc}}$  formalizes the notion of garbage collection by means of a rewrite rule. Such a formalization is required to precisely characterize the semantics of weak references. However, the inclusion of a garbage-collection rewrite-rule in a language with *weak references* introduces non-deterministic evaluation, even if the parameter-passing mechanism is deterministic (call-by-value in our case). This raises the question of confluence for our rewrite system. We discuss natural restrictions under which our rewrite system is confluent, thus guaranteeing uniqueness of program result. We define conditions that allow other garbage collection algorithms to co-exist with our semantics of weak references. We also introduce a polymorphic type system to prove the absence of erroneous program behavior (i.e., the absence of “stuck evaluation”) and a corresponding type inference algorithm. We prove the type system sound and the inference algorithm sound and complete.

## 1 Introduction

### Motivation Behind Weak References

Weak references are references to an object that is not followed by the pointer tracer when garbage collection is called. That is, a weak reference cannot prevent the object it references from being garbage collected. Most language implementations that support weak references (SMLofNJ, Hugs-GHC, OCaml, Java) allow a weak reference to be dereferenced, determining whether the object pointed to has been garbage collected and, if not, what the object is [SML, Mosa, Mosb, Hug98, OCa, Sun].

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 [CMP00]. 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!

As a more concrete example, suppose we wish to hash-cons several lists before we physically store these lists on the heap. The goal of hash-consing is to ensure that structurally equal data values share the same physical storage (therefore reducing memory consumption as well as providing other benefits like quick equality checking). To do this we maintain a hash table which is indexed by each unique list seen thus far and whose entries contain a pointer to the memory location holding the cons cell that corresponds to this list. When we hash-cons a new list, we first check the hash table to determine if each cons cell of the list has been seen before. If so, then the cons cell's memory representation is simply that which the entry in the hash table points to. If not, then we add the cons cell to memory and include a new entry in the hash table. In this way we can conserve the space that is used to store our lists.

However, the danger is that by hash-consing many lists we may run out of memory since even when a cons cell is no longer needed by the program it cannot be deallocated by the garbage collector. This is because each cons cell is kept alive by the hash table (the reference count of a cons cell will never reach zero because it will always be referenced by the hash table). To solve this problem we can replace every pointer in the entries of the hash table with a weak reference. Therefore, when our program no longer needs an element, the table will no longer prevent the element from being garbage collected.

## Difficulties Behind Weak References

Defining the operational semantics of weak references has proven to be a challenging task. To quote the Weak signature documentation of Standard ML of New Jersey, “The semantics of weak pointers to immutable data structures in ML is ambiguous.” [SML] 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 [SML] 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
```

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

$$\begin{aligned} \text{weak} & : 'a \rightarrow 'a \text{ weak} \\ \text{strong} & : 'a \text{ weak} \rightarrow 'a \text{ option.} \end{aligned}$$

After evaluation of this expression, `a` is both statically and dynamically dead, 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 Reference Semantics: A Feature, Not A Bug!

Despite the difficulties previously mentioned, the perspective of this report is that the semantics of weak references are a feature, not a bug. We do not believe that the intuitive semantics of weak references are flawed and we certainly realize the ubiquity of weak references in practice. Unfortunately, the intimate connection between weak references and the non-deterministic behavior of garbage collection makes reasoning about weak references very difficult. This work aims to better understand these semantics by proposing a formal system to reason about them, and, in particular, provide a semantics that allows for “uniqueness of program result”. This is not a straightforward exercise (as is evident from the previous subsection).

### Implementation Independent Formalization

We develop a formalization of weak references that abstracts away the implementation and system dependencies in order to remove the above described ambiguity and precisely define the semantics. Of course, another option would be to propose a compiler or runtime-system implementation of weak references that would act without ambiguity, but this option is beyond the scope of the paper. We choose to present a precise and simple description of the semantics of weak references that will enable formal proof of their properties.

As an attempt to analyze weak references independently of their implementation, we propose a formalization in the style of Morrisett, Felleisen, and Harper’s  $\lambda_{gc}$ , which raises the heap of a programming language to the syntactic level allowing various consistency properties of garbage collection to be proved [MFH95].

Our formalization of garbage collection differs from that of  $\lambda_{gc}$  and other formalizations in that we initially include a high-level garbage collection rewrite rule in our basic operational semantics. This decision is a direct result of the fact that garbage collection can influence the result of a program in the presence of weak references (as shown by the earlier example). However, this decision engenders negative consequences. For example, we require the programmer to know a minimum amount about garbage collection in order to understand how programs behave in the presence of weak references. Even more worrisome is that programs no longer evaluate to unique results. To mitigate this problem, we first discuss methods to control the invocation of garbage collection. By doing so we define a calculus with “uniqueness of program evaluation”. However, since it may be unreasonable to ask a programmer to monitor garbage collection in order to reason about his program, we introduce a restricted class of programs that allows programmers to remain oblivious to garbage collection when reasoning about their programs. This method allows for “uniqueness of program results” as opposed to “uniqueness of program evaluation”. We show that we have not restricted the syntax of programs too severely by encoding a “real-world” example of weak references.

It is worth noting that “uniqueness of program evaluation” implies “uniqueness of program result”, trivially. But, in the presence of garbage collection of weak references which is triggered by the runtime system independently of the program, the converse implication may not hold: “uniqueness of result” does not necessarily imply “uniqueness of evaluation”. This is an interesting point in itself. In other functional languages, once a particular evaluation strategy is selected, such as call-by-value, evaluation is unique – and, hence, so is the result. This is not the case for our semantics of weak references: even though we adopt a call-by-value reduction strategy, evaluation is not necessarily unique, i.e., running the same program at a later time may give rise to a different execution sequence.

It should also be pointed out that these semantics for weak references have been developed with those of SML in mind [SML]. However, there are numerous implementations of weak references in other programming languages with different operational semantics (each with advantages and disadvantages). For a survey of

weak reference semantics in several popular programming languages, see Appendix B. We have chosen the semantics of SML as our basis because these are the simplest to present.

## Related Work

There has been very little work devoted to defining the semantics of weak references, despite the fact that numerous languages have implemented them. The semantics of weak references in Standard ML, Haskell, OCaml, and Java are briefly discussed in the following reports [SML, Mosa, Mosb, Hug98, OCa, Sun]. In addition, the implementation of weak references in Haskell is more thoroughly discussed in [JME99]. However, the authors are unaware of any work that formally treats the semantics of weak references as this report does.

## Organization of the Report

In section 2 we present our basic calculus,  $\lambda_{\text{weak}}$ , for formalizing the semantics of weak references. Section 3 discusses variants of  $\lambda_{\text{weak}}$  to allow for deterministic program evaluation and useful programming constructs. Section 4 introduces a restriction on  $\lambda_{\text{weak}}$  that allows all programs to evaluate to unique results without restricting garbage collection. In section 5 we define conditions that allow other garbage collection algorithms to co-exist with our semantics of weak references. Lastly, we introduce a polymorphic type system in section 6 to prove the absence of erroneous program behavior (i.e., the absence of “stuck evaluation”). We present a corresponding type inference algorithm in section 7. Section 8 closes with a summary and a comment on future work. The proofs of all of the theorems can be found in several appendices, starting with Appendix C. Appendix A is an encoding of a real-world program that uses weak references: a weak hash-consing implementation. We show the benefits of this calculus by proving that optimal garbage collection means this program uses optimal space. Appendix B contains a survey of the differing semantics of weak references in many contemporary programming languages.

## Acknowledgments

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

## 2 $\lambda_{\text{weak}}$

This section describes the formalization of the operational semantics of weak references.  $\lambda_{\text{weak}}$  is an appropriately adapted version of  $\lambda_{\text{gc}}$  from Morrisett, Felleisen, and Harper.

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

Below we define the syntax of  $\lambda_{\text{weak}}$ .

### 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) | $e \in \text{Exp}$                                      | $::= x \mid i \mid \langle e_1, e_2 \rangle \mid \pi_1 e \mid \pi_2 e \mid \lambda x. e \mid$<br>$e_1 e_2 \mid \text{let } x = e_1 \text{ in } e_2 \mid$<br>$\text{weak } e \mid \text{ifdead } e_1 e_2 e_3$ |
| (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)     | $A \in \text{Ans}$                                      | $::= \text{letrec } H \text{ in } x$   |

$\lambda_{\text{weak}}$  programs are composed of a heap and an expression. Expressions are variables, integers, pairs of expressions, projections of expressions, abstractions, applications, let-expressions, weak references, and ifdead-expressions. ifdead-expressions are a combinations of a conditional test and a weak reference dereferencing mechanism (this will become clearer once the semantics of  $\lambda_{\text{weak}}$  are defined).

Heaps are finite maps from variables (locations) to heap values, where heap values are a subset of expressions in addition to  $\mathbf{d}$ .  $\mathbf{d}$  is a distinguished constant that is used to replace values during weak object collections (again this will become clearer once the semantics of  $\lambda_{\text{weak}}$  are defined).

## 2.2 Some Definitions

We introduce some useful notation. We use  $\text{Dom}(H)$  and  $\text{Ran}(H)$  to denote the domain and range of heap  $H$ , respectively. We write  $H \uplus \{x \mapsto hv\}$  to extend the function  $H$  with a mapping of  $x$  to  $hv$ , where we assume  $x \notin \text{Dom}(H)$ . We define the subset  $H^w$  of  $H$  by:

$$H^w = \{x \mapsto H(x) \mid H(x) = \text{weak } y \text{ for some } y\}$$

The subset  $H^s$  of  $H$  is  $H^s = H - H^w$ . We define the function  $\text{FV} : \text{Exp} \cup \text{Hval} \cup \text{Heap} \cup \text{Prog} \rightarrow \text{Var}$  to calculate the free-variables of an expression, heap, or program:

$$\begin{aligned}
\text{FV}(i) &= \emptyset \\
\text{FV}(x) &= \{x\} \\
\text{FV}(\langle e_1, e_2 \rangle) &= \text{FV}(e_1) \cup \text{FV}(e_2) \\
\text{FV}(\pi_i e) &= \text{FV}(e) \\
\text{FV}(\lambda x. e) &= \text{FV}(e) - \{x\} \\
\text{FV}(e_1 e_2) &= \text{FV}(e_1) \cup \text{FV}(e_2) \\
\text{FV}(\text{let } x = e_1 \text{ in } e_2) &= \text{FV}(e_1) \cup (\text{FV}(e_2) - \{x\}) \\
\text{FV}(\text{weak } e) &= \text{FV}(e) \\
\text{FV}(\text{ifdead } e_1 e_2 e_3) &= \text{FV}(e_1) \cup \text{FV}(e_2) \cup \text{FV}(e_3) \\
\text{FV}(\mathbf{d}) &= \emptyset \\
\text{FV}(H) &= \cup \{\text{FV}(hv) \mid hv \in \text{Ran}(H)\} - \text{Dom}(H) \\
\text{FV}(\text{letrec } H \text{ in } e) &= (\text{FV}(H) \cup \text{FV}(e)) - \text{Dom}(H)
\end{aligned}$$

We define term-variable substitution as follows:

$$e_1\{x := e_2\} = \begin{cases} e_2 & \text{if } e_1 = x \\ \langle e'_1\{x := e_2\}, e''_1\{x := e_2\} \rangle & \text{if } e_1 = \langle e'_1, e''_1 \rangle \\ \pi_i e'_1\{x := e_2\} & \text{if } e_1 = \pi_i e'_1 \text{ and } i \in \{1, 2\} \\ \lambda y. e'_1\{x := e_2\} & \text{if } e_1 = \lambda y. e'_1 \text{ and } x \neq y \text{ and } y \notin \text{FV}(e_2) \\ e'_1\{x := e_2\} e''_1\{x := e_2\} & \text{if } e_1 = e'_1 e''_1 \\ \mathbf{let } y = e'_1\{x := e_2\} \mathbf{in } e''_1\{x := e_2\} & \text{if } e_1 = (\mathbf{let } y = e'_1 \mathbf{in } e''_1) \text{ and } x \neq y \\ & \text{and } y \notin \text{FV}(e_2) \\ \mathbf{let } y = e'_1\{x := e_2\} \mathbf{in } e''_1 & \text{if } e_1 = (\mathbf{let } y = e'_1 \mathbf{in } e''_1) \text{ and } x = y \\ \mathbf{weak } e'_1\{x := e_2\} & \text{if } e_1 = \mathbf{weak } e'_1 \\ \mathbf{ifdead } e'_1\{x := e_2\} e''_1\{x := e_2\} e'''_1\{x := e_2\} & \text{if } e_1 = \mathbf{ifdead } e'_1 e''_1 e'''_1 \\ e_1 & \text{otherwise} \end{cases}$$

### 2.3 Semantics of $\lambda_{\text{weak}}$

The semantics of  $\lambda_{\text{weak}}$  are given by rewrite rules defined on programs of  $\lambda_{\text{weak}}$ . Formally, each rewrite rule is a binary relation between programs of  $\lambda_{\text{weak}}$ . The algorithm for applying the rewrite rules to a given program  $\text{letrec } H \text{ in } e$  is as follows. If the body of the program,  $e$ , is not a variable, then we decompose  $e$  into an evaluation context  $E$  (an expression with a hole  $[]$  in place of a subexpression) and an instruction  $I$  such that  $e = E[I]$  where  $E[I]$  denotes replacing the hole in  $E$  with the instruction  $I$ . Morrisett, Felleisen, and Harper note that the evaluation context corresponds to the control state and the instruction roughly corresponds to the program counter of a program. The instruction uniquely determines the rewrite rule that applies to the program and yields a new heap  $H'$  and program body  $e'$ . The resulting program then has the form:  $\text{letrec } H' \text{ in } E[e']$ .

Below we define the evaluation contexts and instructions.

#### Evaluation Contexts and Instruction Expressions:

$$\begin{aligned} \text{(contexts)} \quad E \in \text{Ctxt} & ::= [] \mid \langle E, e \rangle \mid \langle x, E \rangle \mid \pi_i E \mid E e \mid x E \mid \\ & \quad \mathbf{let } x = E \mathbf{in } e \mid \mathbf{weak } E \mid \\ & \quad \mathbf{ifdead } E e_1 e_2 \\ \text{(instruction)} \quad I \in \text{Instr} & ::= hv \mid \pi_i x \mid x y \mid \mathbf{let } x = y \mathbf{in } e \mid \\ & \quad \mathbf{ifdead } x e_1 e_2 \end{aligned}$$

The evaluation contexts are designed to apply a left-to-right, call-by-value evaluation strategy. We next define the operational semantics of  $\lambda_{\text{weak}}$ .

#### 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  $H(x) = \lambda z. e$
- (let-in)  $\text{letrec } H \text{ in } E[\text{let } x = y \text{ in } e] \xrightarrow{\text{let-in}} \text{letrec } H \text{ in } E[e\{x := y\}]$
- (ifdead)  $\text{letrec } H \text{ in } E[\text{ifdead } x e_1 e_2] \xrightarrow{\text{ifdead}} \begin{cases} \text{letrec } H \text{ in } E[e_2 w] & \text{if } H(x) = \text{weak } w \\ \text{letrec } H \text{ in } E[e_1] & \text{if } H(x) = \mathbf{d} \end{cases}$

At this point  $\text{let } x = e_1 \text{ in } e_2$  is simply syntactic sugar for  $e_2\{x := e_1\}$ . In section 3 it plays a more meaningful role. Notice that dereferencing a reference is explicit within this calculus. This is not the case for  $\lambda_{\text{gc}}$ . The main motivation behind this decision is that implicit dereferencing is not widely used in practice and the `ifdead` construct gives the programmer a way to test whether a weakly referenced object has been garbage collected, a capability that is essential in practical uses of weak references.

The following notation is used to reduce a program to normal form by a given set of rewrite rules.

**Definition 2.1 (Full Evaluation).** For any  $\lambda_{\text{weak}}$  programs  $P, P'$ , and set of rewrite rules  $R$  we take  $P \Downarrow_R P'$  to mean that  $P \xrightarrow{R}^* P'$  and  $P'$  is not reducible with respect to  $R$ .  $\square$

Before completing our discussion of the rewrite rules of  $\lambda_{\text{weak}}$  we add one additional rule. In order to do so we must give two auxiliary rules, although neither of these are part of the operational semantics of  $\lambda_{\text{weak}}$  themselves. The rules deal with the garbage collection of heap values and weakly referenced objects.

- (gc)  $\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e \xrightarrow{\text{gc}} \text{letrec } H \text{ in } e$   
provided  $x \notin \text{FV}(\text{letrec } H^s \text{ in } e)$
- (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  $y \notin \text{Dom}(H)$

The (gc) rule garbage collects strong bindings from the heap if the right-hand side of the binding is not reachable from the program when we ignore all weak bindings. This rule is essentially the same as (fv) from Morrisett, Felleisen, and Harper's  $\lambda_{\text{gc}}$ , except that this rule collects only one binding at a time.

The (weak-gc) rule garbage collects weak bindings from the heap. Observe that weak bindings are not immediately discarded, but instead bound to the distinguished constant  $\mathbf{d}$ . At a later stage of program execution, it is possible that bindings to  $\mathbf{d}$  are garbage collected by the rule (gc). We have chosen to formulate (weak-gc) in this way to avoid dangling references in the case that a reachable weak binding is collected. The side-condition of (weak-gc) allows a weak binding to be collected if the bound object (always a variable) does not occur in the strong domain of the remaining heap.

The informal explanation for how weak references are handled is that if nothing keeps alive the object a weak reference points to, then the object can be collected. Hence, one way to interpret the side-condition of (weak-gc) is that a weakly referenced object is alive iff it occurs in the heap.

Finally, we define the additional rewrite rule of  $\lambda_{\text{weak}}$ .

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

Notice that (garb) applies (gc) as many times as possible to the program and then applied (weak-gc) as many times as possible to this result. Therefore, this rule garbage collects all possible garbage bindings from the heap during each invocation.

We name  $R$  the collection of all rewrite rules:  $R = \{\text{alloc}, \pi_1, \pi_2, \text{app}, \text{let-in}, \text{ifdead}, \text{garb}\}$ . An ill-formed  $\lambda_{\text{weak}}$  program is one that is not reducible by the rewrite rules of  $R - \{\text{garb}\}$ .

**Definition 2.2 (Struck Programs).** Any  $\lambda_{\text{weak}}$  program that is not an answer and is not reducible with respect to  $R - \{\text{garb}\}$  is *stuck*. We say that a  $\lambda_{\text{weak}}$  program is *well-formed* if it does not reduce to a stuck program.  $\square$

A program reduces to an answer, or diverges, or becomes stuck.<sup>1</sup> We describe the form of a stuck program.

**Lemma 2.3 (Forms of Stuck Programs).** A  $\lambda_{\text{weak}}$  program is stuck if and only if it is of one of the following forms:

$$\begin{aligned} \text{letrec } H \text{ in } E[\pi_i x] & \quad \text{where } x \notin \text{Dom}(H) \text{ or } H(x) \neq \langle x_1, x_2 \rangle \\ \text{letrec } H \text{ in } E[x y] & \quad \text{where } x \notin \text{Dom}(H) \text{ or } H(x) \neq \lambda z.e \\ \text{letrec } H \text{ in } E[\text{ifdead } x e_1 e_2] & \quad \text{where } x \notin \text{Dom}(H) \text{ or } (H(x) \neq \text{weak } w \text{ and } H(x) \neq \text{d}) \quad \square \end{aligned}$$

The next example shows the rewrite rules at work as well as the non-confluence of  $\lambda_{\text{weak}}$ .

EXAMPLE 2.4.

$$\begin{aligned} & \text{letrec } \{ \} \text{ in } (\lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y)) \langle 5, 6 \rangle \\ \xrightarrow{\text{alloc}} & \text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y) \} \text{ in } a \langle 5, 6 \rangle \\ \xrightarrow{\text{alloc}} & \text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5 \} \text{ in } a \langle b, 6 \rangle \\ \xrightarrow{\text{alloc}} & \text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6 \} \text{ in } a \langle b, c \rangle \\ \xrightarrow{\text{alloc}} & \text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle \} \\ & \text{in } a e \\ \xrightarrow{\text{app}} & \text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle \} \\ & \text{in ifdead } (\text{weak } e) 0 (\lambda y. \pi_1 y) \\ \xrightarrow{\text{alloc}} & \text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle, f \mapsto \text{weak } e \} \\ & \text{in ifdead } f 0 (\lambda y. \pi_1 y) \\ \xrightarrow{\text{ifdead}} & \text{letrec } \{ a \mapsto \lambda x. \text{ifdead } (\text{weak } x) 0 (\lambda y. \pi_1 y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle, f \mapsto \text{weak } e \} \\ & \text{in } (\lambda y. \pi_1 y) e \\ & \longrightarrow \dots \xrightarrow{\text{garb}} \text{letrec } \{ b \mapsto 5 \} \text{ in } b \\ \text{or} & \\ \xrightarrow{\text{garb}} & \text{letrec } \{ f \mapsto \text{d} \} \text{ in ifdead } f 0 (\lambda y. \pi_1 y) \\ & \longrightarrow \dots \xrightarrow{\text{garb}} \text{letrec } \{ g \mapsto 0 \} \text{ in } g \end{aligned}$$

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

Notice that  $\lambda_{\text{weak}}$  is not confluent because  $(\text{garb})$  can apply at any time in the computation of the program. An obvious way to regain confluence is to restrict the use of  $(\text{garb})$ .

### 3 Strategies For Using Garbage Collection

This section, as section 4, aims to recover unique results of  $\lambda_{\text{weak}}$  programs. In this section we investigate different ways of triggering garbage collection. Unlike conventional formalizations of garbage collection,

<sup>1</sup>An answer of the form  $(\text{letrec } H \text{ in } x)$  is not necessarily a normalized expression, if  $H$  contains bindings that do not contribute to the value of  $x$ . All such bindings are garbage collected by finitely many uses of  $(\text{garb})$ , thus producing a normalized answer.

we precisely define when the garbage collection rule will be used in order to have unique answers for the programs.

In 3.1 and 3.2 we restrict the rewrite rules of  $\lambda_{\text{weak}}$  so that program evaluation becomes deterministic. This restriction makes our system deterministic but it also imposes a greater burden on the programmer. Now the programmer is required to know both how garbage collection works and also when garbage collection is used. This a troublesome requirement that will be addressed in section 4.

In 3.3 we do not address the non-determinism of  $\lambda_{\text{weak}}$ , rather we accept the non-deterministic evaluation of  $\lambda_{\text{weak}}$  and provide a facility for the programmer to prevent garbage collection of specified objects.

### 3.1 Capacity-driven Garbage Collection

One strategy for using garbage collection is to monitor the size of the heap and trigger garbage collection when the program exceeds the heap's maximal capacity. This type of garbage collection trigger closely resembles the way that garbage collection is often implemented in practice. For example, SMLofNJ also initiates garbage collection when the heap has reached a certain size, although the process is much more complicated in reality.

We formalize this strategy by parameterizing  $\lambda_{\text{weak}}$  with a maximal size,  $k$ , of the heap. We call this system  $\lambda_{\text{weak}}(k)$ . The syntax, evaluation contexts, and instruction expressions of  $\lambda_{\text{weak}}(k)$  are exactly the same as those of  $\lambda_{\text{weak}}$  but we must adjust the rewrite rules to restrict the application of (garb).

One additional piece of notation: we use  $|H|$  to denote the number of bindings in heap  $H$ . More precisely,  $|H| = |\text{Dom}(H)|$ . When defining the operational semantics of  $\lambda_{\text{weak}}(k)$  we do *not* include:

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

in the *basic* rewrite rules. Instead (garb) is used as an *auxiliary* rule.

#### Rewrite Rules:

$$\begin{aligned} \text{(alloc)} \quad & \text{letrec } H \text{ in } E[hv] \xrightarrow{\text{alloc}} \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } E[x] \\ & \text{provided } |H| < k \text{ and } x \text{ is a fresh variable} \\ \text{(alloc-gc)} \quad & \text{letrec } H \text{ in } E[hv] \xrightarrow{\text{alloc-gc}} \text{letrec } H' \uplus \{x \mapsto hv\} \text{ in } E[x] \\ & \text{provided } |H| = k, \text{letrec } H \text{ in } E[hv] \xrightarrow{\text{garb}} \text{letrec } H' \text{ in } E[hv] \text{ and } |H'| < k \\ \text{(proj)} \quad & \text{letrec } H \text{ in } E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H \text{ in } E[x_i] \\ & \text{provided } H(x) = \langle x_1, x_2 \rangle \text{ and } i \in \{1, 2\} \\ \text{(app)} \quad & \text{letrec } H \text{ in } E[x y] \xrightarrow{\text{app}} \text{letrec } H \text{ in } E[e\{z := y\}] \\ & \text{provided } H(x) = \lambda z.e \\ \text{(let-in)} \quad & \text{letrec } H \text{ in } E[\text{let } x = y \text{ in } e] \xrightarrow{\text{let-in}} \text{letrec } H \text{ in } E[e\{x := y\}] \\ \text{(ifdead)} \quad & \text{letrec } H \text{ in } E[\text{ifdead } x e_1 e_2] \xrightarrow{\text{ifdead}} \begin{cases} \text{letrec } H \text{ in } E[e_2 w] & \text{if } H(x) = \text{weak } w \\ \text{letrec } H \text{ in } E[e_1] & \text{if } H(x) = \mathbf{d} \end{cases} \end{aligned}$$

Note this garbage collection strategy requires the programmer to be aware of the memory consumption needs of their program. That is, if the programmer want to know exactly when a weakly referenced object has been garbage collected then he must be able to monitor the size of heap in order to determine when garbage collection will occur. This is not a straightforward task. In addition, since small changes to a program can often cause large changes in the heap size, this strategy may prove troublesome to real-world

programming. Note also that the same program evaluated with different capacities may produce different results. We now have:

$$R = \{\text{alloc}, \text{alloc-gc}, \pi_1, \pi_2, \text{app}, \text{let-in}, \text{let-in-gc}, \text{ifdead}\}.$$

As a result of these changes we must augment the forms of stuck program since a  $\lambda_{\text{weak}}(k)$  program can become stuck when its heap reaches the maximum capacity and garbage collection doesn't reduce its size.

**Lemma 3.1 (Forms of Stuck Programs).** *A  $\lambda_{\text{weak}}(k)$  program is stuck iff it is of one of the following forms:*

$$\begin{aligned} \text{letrec } H \text{ in } E[\pi_i x] & \quad \text{where } x \notin \text{Dom}(H) \text{ or } H(x) \neq \langle x_1, x_2 \rangle \\ \text{letrec } H \text{ in } E[x y] & \quad \text{where } x \notin \text{Dom}(H) \text{ or } H(x) \neq \lambda z.e \\ \text{letrec } H \text{ in } E[hv] & \quad \text{where } |H| = k \text{ and } \text{letrec } H \text{ in } E[hv] \xrightarrow{\text{garb}} \text{letrec } H \text{ in } E[hv] \\ \text{letrec } H \text{ in } E[\text{ifdead } x e_1 e_2] & \quad \text{where } x \notin \text{Dom}(H) \text{ or } (H(x) \neq \text{weak } w \text{ and } H(x) \neq \text{d}) \quad \square \end{aligned}$$

As a result of the preceding restrictions on garbage collection the following theorem holds for  $\lambda_{\text{weak}}(k)$ .

**Theorem 3.2 (Deterministic Evaluation).** *For any  $\lambda_{\text{weak}}(k)$  program  $M$ , if  $M \xrightarrow{R} N$  for some  $\lambda_{\text{weak}}(k)$  program  $N$ , then  $N$  is uniquely defined.  $\square$*

*Proof Sketch.* By straightforward examination of the rewrite rules of  $\lambda_{\text{weak}}(k)$ .  $\square$

### 3.1.1 Comparing SMLofNJ and $\lambda_{\text{weak}}(k)$

The following examples compare the weak reference semantics of  $\lambda_{\text{weak}}(k)$  with those SMLofNJ in an effort to evaluate the implementation of weak reference in SMLofNJ. We choose to show only examples that illustrate differences between the garbage collection of SML and that of  $\lambda_{\text{weak}}(k)$ . It is not difficult to find examples in which the semantics of SML and  $\lambda_{\text{weak}}(k)$  behave identically.

EXAMPLE 3.3. Consider the following SML program:

```
let val x = 3
in let val w = weak(x)
    in (w,5) end
end
```

The SMLofNJ compiler does not garbage collect  $\mathbf{x}$ . This program is encoded in  $\lambda_{\text{weak}}(k)$  as follows:

```
letrec {} in let x = 3 in let w = weak x in (w,5)
```

Unlike the semantics of SMLofNJ, the rewrite rules of  $\lambda_{\text{weak}}(3)$  allow  $x$  to be garbage collected. If  $\mathbf{x}$  is bound to  $\mathbf{ref } 3$  rather than  $3$  in this example, then SMLofNJ will allow  $\mathbf{x}$  to be garbage collected. This elucidates an obvious ambiguity in SMLofNJ's semantics of weak references (according to SMLofNJ's documentation, weak references are not supposed to prevent an object from being garbage collected).  $\square$

EXAMPLE 3.4. Consider the following SML program:

```

let val x = (3,4)
in let fun f y = let val z = x in y end
    in let val w = weak(x)
        in (w,f) end
    end
end
end

```

The SMLofNJ compiler does garbage collect  $\mathbf{x}$ . This program is encoded in  $\lambda_{\text{weak}}(k)$  as follows:

```

letrec {} in let x = ⟨3,4⟩ in
let f = (λy.let z = x in y) in let w = weak x in ⟨w, f⟩

```

Unlike the semantics of SMLofNJ, the rewrite rules of  $\lambda_{\text{weak}}(k)$  do not garbage collect  $x$  for any  $k$ .

Although, SMLofNJ is able to realize that  $\mathbf{x}$  will not be used and garbage collects it, this action disrupts the semantics of weak references. From the program syntax (code optimizations aside),  $\mathbf{x}$  is a reachable object and therefore alive. Thus, the programmer would expect  $\text{strong}(\mathbf{w})$  to return  $\text{SOME}(\mathbf{x})$ , but this is not the case.  $\square$

### 3.2 Scope-driven Garbage Collection

Another strategy for using garbage collection is to trigger collection every time the result of a rewrite rule produces a program that has exited a lexical scope. For example, consider the following SML program:

```

let val a =
  let val b = 5
    in b end
in a end

```

In this example, when the body of the outer let-expression is evaluated, the bindings  $b = 5$  becomes unreachable because it is out of scope and is therefore garbage. Since all bindings defined within a scope become unreachable after the exit of that lexical scope, this is natural time to perform garbage collection.

Notice that in  $\lambda_{\text{weak}}$  an expression of the form  $\text{let } x = e_1 \text{ in } e_2$  is simply syntactic sugar for  $(\lambda x.e_2) e_1$ . However, in this section we define a calculus called  $\lambda_{\text{weak-scope}}$  in which the **let-in** construct is used to trigger garbage collection. This provides the programmer with the ability to control the points of the program execution at which garbage collection is triggered. However, it also requires the programmer to insert enough **let-in**'s into their program to prevent a memory overflow.

The syntax, evaluation contexts and instruction expressions of  $\lambda_{\text{weak-scope}}$  are identical to  $\lambda_{\text{weak}}$ . But the rewrite rules are slightly changed. In  $\lambda_{\text{weak-scope}}$  the rule:

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

is *not* part of the *basic* rewrite rules of the language. Instead it is used as an *auxiliary* rule by (**let-in**).

#### 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
- (proj)  $\text{letrec } H \text{ in } E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H \text{ in } E[x_i]$   
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  $H(x) = \lambda z. e$
- (let-in)  $\text{letrec } H \text{ in } E[\text{let } x = y \text{ in } e] \xrightarrow{\text{let-in}} \text{letrec } H' \text{ in } E[e\{x := y\}]$   
 $(\text{letrec } H \text{ in } E[\text{let } x = y \text{ in } e]) \xrightarrow{\text{garb}} \text{letrec } H' \text{ in } E[\text{let } x = y \text{ in } e]$
- (ifdead)  $\text{letrec } H \text{ in } E[\text{ifdead } x \ e_1 \ e_2] \xrightarrow{\text{ifdead}} \begin{cases} \text{letrec } H \text{ in } E[e_2 w] & \text{if } H(x) = \text{weak } w \\ \text{letrec } H \text{ in } E[e_1] & \text{if } H(x) = \text{d} \end{cases}$

We now have:  $R = \{\text{alloc}, \pi_1, \pi_2, \text{app}, \text{let-in}, \text{ifdead}\}$ .

Consider the following  $\lambda_{\text{weak-scope}}$  program:  $(\text{letrec } H \text{ in } (\lambda x. (\lambda y. (\lambda z. z) \langle 5, y \rangle) x) 6)$ . In  $\lambda_{\text{weak}}$  this program is equivalent to either:  $(\text{letrec } H \text{ in } \text{let } x = 6 \text{ in } \text{let } y = x \text{ in } \text{let } z = \langle 5, y \rangle \text{ in } z)$  or  $(\text{letrec } H \text{ in } \text{let } x = 6 \text{ in } (\lambda y. (\lambda z. z) \langle 5, y \rangle) x)$ . But in  $\lambda_{\text{weak-scope}}$  the execution of the first program will use garbage collection 3 times while the execution of the second program will only trigger garbage collection once.

REMARK 3.5. Notice that scope-driven garbage collection resembles region-based memory management in the sense that both techniques invoke garbage collection after leaving a lexical scope [TT97].  $\square$

The form of a stuck  $\lambda_{\text{weak-scope}}$  program is the same as that of  $\lambda_{\text{weak}}$ . The next theorem holds for  $\lambda_{\text{weak-scope}}$ .

**Theorem 3.6 (Deterministic Evaluation).** *For any  $\lambda_{\text{weak-scope}}$  program  $M$ , if  $M \xrightarrow{R} N$  for some  $\lambda_{\text{weak-scope}}$  program  $N$ , then  $N$  is uniquely defined.*  $\square$

*Proof Sketch.* By straightforward examination of the rewrite rules of  $\lambda_{\text{weak-scope}}$ .  $\square$

### 3.2.1 Comparing SMLofNJ and $\lambda_{\text{weak-scope}}$

In this section we use similar examples to those from section 3.1.1 to compare the weak reference semantics of  $\lambda_{\text{weak-scope}}$  with those SMLofNJ in an effort to evaluate the implementation of weak reference in SMLofNJ. The comments of section 3.1.1 also apply to these examples so we do not elaborate on the difference between the semantics of SMLofNJ and  $\lambda_{\text{weak-scope}}$  here.

EXAMPLE 3.7. Consider the following SML program:

```
let val x = 3
in let val w = weak(x)
    in let val z = (w,5)
        in z end
    end
end
```

The SMLofNJ compiler does not garbage collect  $x$ . This program is encoded in  $\lambda_{\text{weak-scope}}$  as follows:

```
letrec {} in let x = 3 in let w = weak x in let z = (w,5) in z
```

Unlike the semantics of SMLofNJ, the rewrite rules of  $\lambda_{\text{weak-scope}}$  allow  $x$  to be garbage collected.  $\square$

EXAMPLE 3.8. Consider the following SML program:

```
let val x = (3,4)
in let fun f y = let val z = x in y end
    in let val w = weak(x)
        in (w,f) end
    end
end
```

The SMLofNJ compiler does garbage collect  $x$ . This program is encoded in  $\lambda_{\text{weak-scope}}$  as follows:

```
letrec {} in let x = ⟨3,4⟩ in
let f = (λy.let z = x in y) in let w = weak x in ⟨w, f⟩
```

Unlike the semantics of SMLofNJ, the rewrite rules of  $\lambda_{\text{weak-scope}}$  do not garbage collect  $x$ .  $\square$

### 3.3 Preventing Garbage Collection of Specified Objects

The two previous subsections introduced methods for regaining uniqueness of answers. However, both methods introduce further difficulties ( $\lambda_{\text{weak}}(k)$  requires programmers to monitor the size of the heap,  $\lambda_{\text{weak-scope}}$  requires programmers to insert enough **let-in**'s to prevent memory over-flow). As an alternative, this section defines a syntactic utility that allows programmers to designate heap objects that are never eligible for garbage collection. We call this calculus  $\lambda_{\text{weak-prevent}}$ . The syntax of  $\lambda_{\text{weak-prevent}}$  is that of  $\lambda_{\text{weak}}$  with the addition of the following:

(expressions)  $e \in \text{Exp} ::= \dots \mid \text{prev } e$   
(heap values)  $hv \in \text{Hval} ::= \dots \mid \text{prev } hv$

We also make the following addition to our definition of FV:  $\text{FV}(\text{prev } e) = \text{FV}(e)$ . We also modify the definition of  $\xrightarrow{\text{gc}}$  slightly, but leave the definition of  $\xrightarrow{\text{weak-gc}}$  unchanged.

(gc)  $\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e \xrightarrow{\text{gc}} \text{letrec } H \text{ in } e$   
provided  $x \notin \text{FV}(\text{letrec } H^s \text{ in } e)$  and  $hv \neq \text{prev } hv'$  for some  $hv'$   
(weak-gc)  $\text{letrec } H \uplus \{x \mapsto \text{weak } y\} \text{ in } e \xrightarrow{\text{weak-gc}} \text{letrec } H \uplus \{x \mapsto d\} \text{ in } e$   
provided  $y \notin \text{Dom}(H)$

Notice that according to the definition of  $H^s$ , all bindings of  $H$  of the form  $x \mapsto \text{prev } hv$  are included in  $H^s$ , since such a binding is not included in  $H^w$ .

The rewrite rules of  $\lambda_{\text{weak}}$  are also the rewrites of  $\lambda_{\text{weak-prevent}}$ . However, there is a trivial difference between the rules of  $\lambda_{\text{weak}}$  and those of  $\lambda_{\text{weak-prevent}}$ . Any side-condition containing  $H(x) = hv$ , for any  $hv$ , in  $\lambda_{\text{weak}}$  is written  $H(x) = hv$  or  $H(x) = \text{prev } hv$  in  $\lambda_{\text{weak-prevent}}$ . Since the rewrite rules of  $\lambda_{\text{weak-prevent}}$  are identical besides this difference, we do not re-define them here. Consider the following example.

EXAMPLE 3.9.

$$\begin{array}{l}
\text{letrec } \{\} \text{ in } (\lambda x.\text{ifdead } (\text{prev weak } x) 0 \lambda y.\pi_1 y) \langle 3, 5 \rangle \\
\frac{\text{alloc}}{\longrightarrow} \text{letrec } \{a \mapsto \lambda x.\text{ifdead } (\text{prev weak } x) 0 \lambda y.\pi_1 y\} \text{ in } a \langle 3, 5 \rangle \\
\frac{\text{alloc}}{\longrightarrow} \text{letrec } \{a \mapsto \lambda x.\text{ifdead } (\text{prev weak } x) 0 \lambda y.\pi_1 y, b \mapsto 3\} \text{ in } a \langle b, 5 \rangle \\
\frac{\text{alloc}}{\longrightarrow} \text{letrec } \{a \mapsto \lambda x.\text{ifdead } (\text{prev weak } x) 0 \lambda y.\pi_1 y, b \mapsto 3, c \mapsto 5\} \text{ in } a \langle b, c \rangle \\
\frac{\text{alloc}}{\longrightarrow} \text{letrec } \{a \mapsto \lambda x.\text{ifdead } (\text{prev weak } x) 0 \lambda y.\pi_1 y, b \mapsto 3, c \mapsto 5, e \mapsto \langle b, c \rangle\} \\
\text{in } a e \\
\frac{\text{app}}{\longrightarrow} \text{letrec } \{a \mapsto \lambda x.\text{ifdead } (\text{prev weak } x) 0 \lambda y.\pi_1 y, b \mapsto 3, c \mapsto 5, e \mapsto \langle b, c \rangle\} \\
\text{in ifdead } (\text{prev weak } e) 0 \lambda y.\pi_1 y \\
\frac{\text{alloc}}{\longrightarrow} \text{letrec } \{a \mapsto \lambda x.\text{ifdead } (\text{prev weak } x) 0 \lambda y.\pi_1 y, b \mapsto 3, c \mapsto 5, e \mapsto \langle b, c \rangle, \\
f \mapsto \text{prev weak } e\} \text{ in ifdead } f 0 \lambda y.\pi_1 y \\
\frac{\text{garb}}{\longrightarrow} \text{letrec } \{b \mapsto 3, c \mapsto 5, e \mapsto \langle b, c \rangle, f \mapsto \text{prev weak } e\} \text{ in ifdead } f 0 \lambda y.\pi_1 y \\
\frac{\text{ifdead}}{\longrightarrow} \text{letrec } \{b \mapsto 3, c \mapsto 5, e \mapsto \langle b, c \rangle, f \mapsto \text{prev weak } e\} \text{ in } (\lambda y.\pi_1 y) e \\
\frac{\text{alloc}}{\longrightarrow} \text{letrec } \{b \mapsto 3, c \mapsto 5, e \mapsto \langle b, c \rangle, f \mapsto \text{prev weak } e, h \mapsto \lambda y.\pi_1 y\} \text{ in } h e \\
\frac{\text{app}}{\longrightarrow} \text{letrec } \{b \mapsto 3, c \mapsto 5, e \mapsto \langle b, c \rangle, f \mapsto \text{prev weak } e, h \mapsto \lambda y.\pi_1 y\} \text{ in } \pi_1 e \\
\frac{\text{garb}}{\longrightarrow} \text{letrec } \{b \mapsto 3, c \mapsto 5, e \mapsto \langle b, c \rangle, f \mapsto \text{prev weak } e\} \text{ in } \pi_1 e \\
\frac{\pi_1}{\longrightarrow} \text{letrec } \{b \mapsto 3, c \mapsto 5, e \mapsto \langle b, c \rangle, f \mapsto \text{prev weak } e\} \text{ in } b
\end{array}$$

□

## 4 Programs That Are Oblivious to Garbage Collection

It is possible to restrict the syntax of  $\lambda_{\text{weak}}$  in order to prevent garbage collection from affecting the final result of a program. A person programming within this restriction can remain entirely oblivious to the process of garbage collection. Unlike the previous section, “uniqueness of program result” is here achieved not by “uniqueness of program evaluation”, which still required programmer’s awareness of garbage collection in one way or another. Rather, we achieve “uniqueness of program result” by restricting the syntax of  $\lambda_{\text{weak}}$  so that the same program always produces the same result even though it may evaluate by different sequences of rewrite rules. As motivation, consider the following small example:

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

It is not difficult to see that this program will always evaluate to the same result. Notice the fact that this program has a unique result does not obviate the purpose of weak references. Such a program can still make use of weak references to reduce its memory consumption. Also, notice the duplication of the expression the weak-reference points to does not cause extra memory consumption, since this expression is merely in the syntax of the program and will not be allocated to heap unless the object the weak-reference points to has been garbage collected. We call a program that evaluates to a unique result, such as this one, “well-behaved”.

The next example shows another program, which is a little more difficult to identify as “well-behaved”:

$$\text{letrec } \{\} \text{ in ifdead } (\pi_1 \langle \text{weak } 5, 3 \rangle) ((\lambda x.x) 5) (\lambda x.x)$$

In general, determining whether an ifdead-expression of the form  $(\text{ifdead } e_1 (e_2 e_3) e_2)$  is “well-behaved” amounts to determining whether  $e_1$  and  $(\text{weak } e_3)$  evaluate to the same result. Since  $\lambda_{\text{weak}}$  encodes the entire untyped  $\lambda$ -calculus, this equivalence is undecidable. Therefore, we cannot hope to have a syntactic characterization of all “well-behaved”  $\lambda_{\text{weak}}$  programs, i.e., an effective procedure that will determine whether an arbitrary  $\lambda_{\text{weak}}$  program always evaluates to the same result. However, we can describe a proper subset

of “well-behaved” programs and argue that this subset is large enough to include many realistic usages of weak references (such as an implementation of hash-consing for lists).

We call our proper subset of “well-behaved” programs the set of “gc-oblivious” programs (for lack of a better name), in Definition 4.3.

We introduce a restricted set of expressions,  $\text{Exp}^*$ , defined simultaneously with a set of pairs of expressions,  $\text{ExpPair}$ . The intended meaning of a pair  $(e_1, e_2) \in \text{ExpPair}$ , is this: *If an expression of the form  $(\text{ifdead } e_1 (e_3 e_2) e_3)$  converges then there is a heap value  $hv$  such that  $e_1$  converges to  $(\text{weak } hv)$  and  $e_2$  converges to  $hv$ .*

It is possible to give a definition of  $\text{Exp}^*$  and  $\text{ExpPair}$  in extended BNF notation in the style of Section 2.1. It is easier, however, to define them using structural rules as follows:

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

By structural induction, it is readily checked that  $\text{Exp}^* \subset \text{Exp}$  and  $\text{ExpPair} \subset \text{Exp} \times \text{Exp}$ . There are two differences between  $\text{Exp}$  and  $\text{Exp}^*$ : (1) the omission of let-declarations in  $\text{Exp}^*$ , and (2) the restriction we impose on  $\text{ifdead}$ -expressions.

REMARK 4.1. An  $\text{ifdead}$ -expression in  $\text{Exp}^*$  is always of the form

$$\text{ifdead } e_1 (e_3 e_2) e_3 \quad \text{where } (e_1, e_2) \in \text{ExpPair}$$

The evaluation of an  $\text{Exp}^*$  expression  $e$  does not necessarily produce another  $\text{Exp}^*$  expression  $e'$ , because  $e'$  may contain an  $\text{ifdead}$ -expression  $(\text{ifdead } e_1 (e_3 e_2) e_3)$  where  $(e_1, e_2) \notin \text{ExpPair}$ .  $\square$

It is convenient to make the following definition.

**Definition 4.2 (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 4.3 (GC-Oblivious Programs).** A  $\lambda_{\text{weak}}$  program  $(\text{letrec } \{ \} \text{ in } e)$  is *gc-oblivious* if  $e$  is an expression in the restricted set  $\text{Exp}^*$ .  $\square$

We make the definition of  $\text{Exp}^*$  sufficiently broad to include common usages of weak references, but also simple enough for an efficient procedure to test whether an arbitrary expression is in  $\text{Exp}^*$ .

**Proposition 4.4 (Efficient Test for Membership in  $\text{Exp}^*$  and  $\text{ExpPair}$ ).** *Let  $e_1, e_2$  be arbitrary expressions. There are effective procedures to decide whether  $e_1$  is in the restricted set  $\text{Exp}^*$  and whether  $(e_1, e_2)$  is in  $\text{ExpPair}$ . The procedures run in time which is a (low-degree) polynomial in the sizes of  $e_1, e_2$ .  $\square$*

For a precise statement of the main result of this section we need an additional definition. We define the function  $\text{result}$  which takes a heap  $H$  and an expression  $e$  as arguments, and returns an expression  $e'$  obtained from  $e$  by replacing free variables with their bindings in  $H$ .

**Definition 4.5 (result).**

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

The side condition “ $x \notin \text{Dom}(H)$ ” in the cases for  $e = \lambda x. e'$  and  $e = (\text{let } x = e_1 \text{ in } e_2)$  can always be satisfied by  $\alpha$ -renaming the  $\lambda$ -bound  $x$  and the  $\text{let}$ -bound  $x$ , respectively.  $\square$

The following theorem proved that gc-oblivious programs are well-behaved.

**Theorem 4.6 (GC-Oblivious Programs Are Well-Behaved).** *If  $P$  is gc-oblivious, then one of the following three statements holds:*

1. *Every evaluation of  $P$  diverges, i.e., there is no finite reduction sequence starting from  $P$ .*
2. *Every evaluation of  $P$  becomes stuck, i.e., every reduction of  $P$  terminates at a stuck program.*
3. *Every evaluation of  $P$  terminates and produces the same answer, i.e., if  $P \Downarrow_R (\text{letrec } H_1 \text{ in } x_1)$  and  $P \Downarrow_R (\text{letrec } H_2 \text{ in } x_2)$ , then  $\text{result}(H_1, x_1) = \text{result}(H_2, x_2)$ .  $\square$*

Proofs for this section can be found in Appendix C. For Theorem 4.6 we only prove part 3. We believe that the proofs of parts 1 and 2 will be similar to the proof of part 3 and leave this as future work.

## 4.1 Enlarging the Set of GC-Oblivious Programs

Our definition of  $\text{Exp}^*$  is somewhat *ad hoc*, as it is dictated by no consideration other than encompassing common usages of weak references. It can be extended and changed accordingly. For example, we can parameterize the  $\text{ExpPair}$  relation with  $p \in \{1, 2\}^*$  to obtain a larger set of companion pairs, defined by the rules below. This new relation  $\text{ExpPair}(p)$  allows pairs and projects to occur in companion pairs.

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

The rules for defining  $\text{Exp}^*$ , presented earlier in this section, remain unchanged, except that  $\text{ExpPair}$  now stands for  $\text{ExpPair}(\varepsilon)$  in the premise of the rule to derive ifdead-expressions.

We have parameterized the class of companion pairs  $\text{ExpPair}(p)$  with a “path”  $p \in \{1, 2\}^*$ . For an informal understanding of such a  $p$ , consider an expression  $e$  for which the following rewrite steps can be carried out:

$$\pi_{i_1} e \rightarrow e_1 \quad , \quad \pi_{i_2} e_1 \rightarrow e_2 \quad , \quad \dots \quad , \quad \pi_{i_k} e_{k-1} \rightarrow e_k$$

for some expressions  $e_1, \dots, e_k$ , i.e.,  $(\pi_{i_k} \dots \pi_{i_1} e)$  evaluates to  $e_k$ . Suppose  $e_k$  is of the form  $(\text{weak } e'_k)$ . Then we place the pair  $(e, e')$  in  $\text{ExpPair}(p)$  where  $e'$  is obtained from  $e$  by replacing the component  $e_k$  by  $e'_k$ , and  $p = i_k \dots i_2 i_1$ .

More graphically, if  $e$  is built up from repeated uses of  $\langle \cdot, \cdot \rangle$  and thus represented by a binary parse tree  $T$ , then  $\pi_{i_k} \dots \pi_{i_1} e$  returns the subexpression  $e_k = (\text{weak } e'_k)$  at address  $i_k \dots i_2 i_1$ . The companion expression  $e'$  is obtained from  $e$  by omitting “weak” at address  $i_k \dots i_2 i_1$ .

EXAMPLE 4.7. We give several simple examples of parameterized companion pairs.

1.  $(\text{weak } 5, 5) \in \text{ExpPair}(\varepsilon)$ .
2.  $(\langle \text{weak } 5, 8 \rangle, \langle 5, 8 \rangle) \in \text{ExpPair}(1)$ .
3.  $(\langle \langle 3, \text{weak } 5 \rangle, \langle 3, 5 \rangle \rangle) \in \text{ExpPair}(2)$ .
4.  $(\langle \langle \text{weak } 3, \text{weak } 5 \rangle, \langle 3, 5 \rangle \rangle) \in \text{ExpPair}(1)$  and also  $\in \text{ExpPair}(2)$ ,  
i.e.,  $p$  is not uniquely determined for the same companion pair.
5.  $(\langle \langle \langle 3, \text{weak } 5 \rangle, 8 \rangle, \langle \langle 3, 5 \rangle, 8 \rangle \rangle) \in \text{ExpPair}(12)$ .
6.  $(\pi_1 \langle \langle 3, \text{weak } 5 \rangle, 8 \rangle, \pi_1 \langle \langle 3, 5 \rangle, 8 \rangle) \in \text{ExpPair}(2)$ .
7.  $(\lambda x. \pi_1 \langle \langle 3, \text{weak } x \rangle, 8 \rangle, \lambda x. \pi_1 \langle \langle 3, x \rangle, 8 \rangle) \in \text{ExpPair}(2)$ . □

We can further enlarge the set of companion pairs according to needs. For example, if  $\lambda_{\text{weak}}$  is augmented with conditionals, we can add the following rule:

$$\frac{e \in \text{Exp}^* \quad (e_1, e_2) \in \text{ExpPair}(p) \quad (e_3, e_4) \in \text{ExpPair}(p)}{(\text{if } e \text{ then } e_1 \text{ else } e_3, \text{if } e \text{ then } e_2 \text{ else } e_4) \in \text{ExpPair}(p)}$$

A proof for an efficient test for membership in  $\text{Exp}^*$  and  $\text{ExpPair}(p)$  for any  $p \in \{1, 2\}^*$  is given in Appendix C. We do not prove that gc-obliviousness, with the larger set of companion pairs  $\text{ExpPair}(p)$ , implies well-behavedness – and leave such a proof to the interested reader as a straightforward (though tedious) extension of our proof that gc-obliviousness with  $\text{ExpPair} = \text{ExpPair}(\varepsilon)$  implies well-behavedness.

## 4.2 An Extended Example

We show that gc-oblivious programs are sufficiently inclusive by encoding a common and practical implementation using weak references: *hash-consing*. Our formal semantics for weak references allow us to reason about the behavior of hash-consing precisely.

In appendix A we discuss this example further and demonstrate the utility of our formal semantics by proving that optimal garbage collection implies this example consumes optimal space. The encoding of this program in appendix A maybe easier to read, we have inlined the *hashCons* function from the example in appendix A for the purposes of exposition. We present this example below in a format that contains let-declarations: These are provided for ease of reading. By eliminating the let-declarations we can transform this program into one which is gc-oblivious. The elimination of let-declarations is only done to determine whether this example is gc-oblivious.

EXAMPLE 4.8.

```

letrec {}
in let w0 = weak []
  in let w1 = weak [1]
    in let hash = λl.
      if equals (l, []) then 0
      else if equals (l, [1]) then 1
      else - 1
      in let p =
        if (hash []) = 0
        then ifdead w0 ((λl.⟨l, ⟨weak l, w1⟩⟩) []) (λl.⟨l, ⟨weak l, w1⟩⟩)
        else if (hash []) = 1
          then ifdead w1 ((λl.⟨l, ⟨w0, weak l⟩⟩) [1]) (λl.⟨l, ⟨w0, weak l⟩⟩)
          else ⟨[-1], ⟨w0, w1⟩⟩
          in let p1 =
            if (hash []) = 0
            then ifdead w0 ((λl.⟨l, ⟨l, w1⟩⟩) []) (λl.⟨l, ⟨l, w1⟩⟩)
            else if (hash []) = 1
              then ifdead w1 ((λl.⟨l, ⟨[], weak l⟩⟩) [1]) (λl.⟨l, ⟨[], weak l⟩⟩)
              else ⟨[-1], ⟨w0, w1⟩⟩
              in let p2 =
                if (hash []) = 0
                then ifdead w0 ((λl.⟨l, ⟨weak l, [1]⟩⟩) []) (λl.⟨l, ⟨weak l, [1]⟩⟩)
                else if (hash []) = 1
                  then ifdead w1 ((λl.⟨l, ⟨w0, l⟩⟩) [1]) (λl.⟨l, ⟨w0, l⟩⟩)
                  else ⟨[-1], ⟨w0, w1⟩⟩
                  in let p' =
                    if (hash []) = 0
                    then ifdead (π1 π2 p) ((λl.⟨l, ⟨weak l, w1⟩⟩) π1 π2 p1) (λl.⟨l, ⟨weak l, w1⟩⟩)
                    else if (hash []) = 1
                      then ifdead (π2 π2 p) ((λl.⟨l, ⟨w0, weak l⟩⟩) π2 π2 p2) (λl.⟨l, ⟨w0, weak l⟩⟩)
                      else ⟨[-1], ⟨w0, w1⟩⟩
                      in ⟨π1 p, π1 p'⟩

```

□

This example differs from the example in appendix A in the inclusion of  $p_1, p_2$ . These are included to make the definition of  $p'$  gc-oblivious.

## 5 Additional Strong Garbage Collection

Notice that when defining the operational semantics of  $\lambda_{\text{weak}}$  we have fixed the family of garbage collection algorithms that can be modeled, namely reachability-based garbage collection algorithms. Although other formalizations of garbage collection algorithms in the presence of weak references are possible we have chosen to use a reachability-based algorithm due to its ubiquity in current compiler implementation.

It is not desirable to restrict the semantics of  $\lambda_{\text{weak}}$  to one garbage collection algorithm. In fact, the addition of another garbage collection rewrite rule to our semantics may allow for more garbage to be reclaimed. However, we must take care when augmenting our semantics with an additional garbage collection algorithm because its presence may interfere with the semantics of weak references (we may lose the ability to precisely talk about when a weakly referenced object is garbage). As a result of our desire to allow additional garbage collection and our need to maintain the semantics of weak references when doing so, in this section we describe some restrictions that when placed on an additional garbage collection rewrite rule guarantee that the operational semantics of  $\lambda_{\text{weak}}$  are preserved. First we define these restrictions and then we prove that when such conditions are satisfied the semantics of  $\lambda_{\text{weak}}$  are not violated.

**REMARK 5.1.** In this section we consider allowing an additional garbage collection algorithm that collects strong bindings only. If an additional garbage collection rule were allowed to collect weak bindings then the semantics of  $\lambda_{\text{weak}}$  would certainly be altered.  $\square$

The following definition is adapted from similar definitions by Morrisett, Felleisen, and Harper.

**Definition 5.2 (Kleene Equivalence).**  $(P_1, R_1) \simeq (P_2, R_2)$  means  $P_1 \Downarrow_{R_1} (\text{letrec } H_1 \text{ in } x)$  where  $H_1(x) = i$  if and only if  $P_2 \Downarrow_{R_2} (\text{letrec } H_2 \text{ in } y)$  and  $H_2(y) = i$ .  $\square$

Now we would like to say that any garbage collection rewrite rule can be added to  $\lambda_{\text{weak}}$  as long as it satisfies the following condition:

**Claim 5.3.** We can add  $\xrightarrow{x}$  to the semantics of  $\lambda_{\text{weak}}$ , if for all programs  $P$  where  $P \xrightarrow{x} P'$  for some  $\lambda_{\text{weak}}$  program  $P'$ , we have  $(P, R) \simeq (P', R)$ .  $\square$

However, for this claim to hold, the additional garbage collection rule must respect the semantics of  $\lambda_{\text{weak}}$ . Such concerns are not standard for the designers of garbage collection algorithms. Therefore we give more specific restrictions that must be placed on the additional garbage collection rule and show, operationally, how such restrictions can be enforced.

In the following we will also make use of a rewrite relation defined on triples of heaps, sets of program variables, and heaps respectively:

$$\langle H \uplus \{x \mapsto hv\}, S \uplus \{x\}, H' \rangle \Rightarrow \langle H, S \cup \text{FV}(hv), H' \uplus \{x \mapsto hv\} \rangle.$$

Intuitively, this relation will be used to calculate all those heap bound variables that are utilized by some weak binding in the heap. We would like to prevent the additional garbage collection rule from collecting these bindings since doing so may alter the semantics of weak references. Consider the following example:

$$\text{letrec } \{x \mapsto 5, y \mapsto \langle x, x \rangle, z \mapsto \text{weak } y\} \text{ in } z.$$

In this program the weak binding  $z \mapsto \text{weak } y$  is considered to be garbage if the strong binding  $y \mapsto \langle x, x \rangle$  has been previously collected. But, if the strong binding  $y \mapsto \langle x, x \rangle$  persists then  $y$  remains in the domain of

the heap and therefore  $z$  is alive. So, by allowing an additional garbage collection rule to collect the binding  $x \mapsto 5$  we will alter the result of the evaluation of the program (and actually prevent the system from being confluent if the additional garbage collection rule is allowed to be applied at any time).

As additional notation, we write:

$$\langle H_1, S_1, H'_1 \rangle \Rightarrow_{\text{nf}} \langle H_2, S_2, H'_2 \rangle$$

to mean that

$$\langle H_1, S_1, H'_1 \rangle \Rightarrow^* \langle H_2, S_2, H'_2 \rangle \quad \text{and} \quad \text{Dom}(H_2) \cap S_2 = \emptyset.$$

In order to define the restrictions that we will place on our additional garbage collection rule, we must first define the following transformation function,  $Trans : \text{Prog} \rightarrow \text{Prog}$ . This transformation will be used to convert programs with weak references into similar programs without weak references. The purpose of this transformation will be to provide a mechanism to show that the introduction of weak references does not allow additional garbage collection to occur. Given a  $\lambda_{\text{weak}}$  program  $P \equiv (\text{letrec } H \uplus \{x \mapsto hv\} \text{ in } e)$  we would like to say that the binding  $x \mapsto hv$  is garbage collected in  $P$  only if the same binding  $x \mapsto hv$  is garbage collected in  $Trans(P)$ . However, first we define the function,  $remove : \text{Exp} \rightarrow \text{Exp}$ , that will be used by  $Trans$ .

**Definition 5.4** ( $remove(e)$ ).

$$\begin{aligned} remove(i) &= i \\ remove(x) &= x \\ remove(\langle e_1, e_2 \rangle) &= \langle remove(e_1), remove(e_2) \rangle \\ remove(\pi_i e) &= \pi_i remove(e) \\ remove(\lambda x.e) &= \lambda x.remove(e) \\ remove(e_1 e_2) &= remove(e_1) remove(e_2) \\ remove(\text{let } x = e_1 \text{ in } e_2) &= \text{let } x = remove(e_1) \text{ in } remove(e_2) \\ remove(\text{weak } e) &= remove(e) \\ remove(\text{ifdead } e_1 e_2 e_3) &= \pi_1 \langle remove(e_2), remove(e_3) remove(e_1) \rangle \end{aligned}$$

□

**Definition 5.5** ( $Trans(P)$ ). Let  $Trans(\text{letrec } H \text{ in } e) = \text{letrec } H' \text{ in } e'$  where  $H' = (H - H^w)$  and  $e' = remove(e)$ . □

EXAMPLE 5.6. Consider the  $Trans$  function applied to the following program.

$$\begin{aligned} Trans(\text{letrec } \{x \mapsto 3, y \mapsto 6, z \mapsto \text{weak } y\} \text{ in ifdead } z x (\lambda v.\text{ifdead } ((\lambda x.\text{weak } x) v) 9 \lambda u.u) = \\ \text{letrec } \{x \mapsto 3, y \mapsto 6\} \text{ in } \pi_1 \langle x, (\lambda v.\pi_1 \langle 9, (\lambda u.u) ((\lambda x.x) v) \rangle) z \rangle \end{aligned}$$

□

**Definition 5.7 (Respectful Garbage Collection).** We say that a rule  $\xrightarrow{x}$  is a respectful garbage collection rule if it satisfies the following conditions.

1. For every  $\lambda_{\text{weak}}$  program  $\text{letrec } H \text{ in } e$  we have  $(\text{letrec } H \text{ in } e) \xrightarrow{x} (\text{letrec } H' \text{ in } e)$  for some heap  $H' \subseteq H$ , i.e., every  $\lambda_{\text{weak}}$  program can be reduced using  $\xrightarrow{x}$ .
2. For every  $\lambda_{\text{weak}}$  program  $P$  such that  $\text{weak}$  and  $\text{ifdead}$  do not occur in  $P$  and  $P \xrightarrow{x} P'$  for some  $\lambda_{\text{weak}}$  program  $P'$ , we have  $(P, R) \simeq (P', R)$ .

3. For every  $\lambda_{\text{weak}}$  program  $P$ , if  $\overset{x}{\rightarrow}$  applied to  $P$  garbage collects some binding  $x$ , then  $\overset{x}{\rightarrow}$  applied to  $\text{Trans}(P)$  also garbage collects  $x$ , i.e., the presence of weak references do not enable  $\overset{x}{\rightarrow}$  to collect extra bindings.
4. For all  $\lambda_{\text{weak}}$  programs  $P_1 = \text{letrec } H_1 \text{ in } e_1$  and  $P_2 = \text{letrec } H_2 \text{ in } e_2$  such that  $P_1 \overset{x}{\rightarrow} P_2$ , if  $\langle H_1, \text{Dom}(H_1^w), \emptyset \rangle \Rightarrow_{\text{nf}} \langle H'_1, \emptyset, H''_1 \rangle$  then  $H''_1 \subseteq H_2$ .  $\square$

Notice, by the above definition,  $\overset{\text{garb}}{\rightarrow}$  is not a respectful garbage collection rule. This is because  $\overset{\text{garb}}{\rightarrow}$  can alter weak bindings in the heap.

If the conditions of the preceding definition are met then we can prove the following theorem which is analogous to Claim 5.3.

**Theorem 5.8 (Addition of  $\overset{x}{\rightarrow}$ ).** *If  $\overset{x}{\rightarrow}$  is a respectful garbage collection rule then for all well-formed  $\lambda_{\text{weak}}$  programs  $P$  such that  $P \overset{x}{\rightarrow} P'$  for some  $\lambda_{\text{weak}}$  program  $P'$  we have  $(P, \mathbf{R}) \simeq (P', \mathbf{R})$ .*  $\square$

The proof of this theorem and all required lemmas can be found in the appendix.

Now let us inspect the conditions in the definition of a respectful garbage collection rule. Notice that the first and second conditions are standard results that must be shown to hold for any legitimate garbage collection algorithm (the second condition is analogous to saying that the garbage collection algorithm works correctly in a system without weak references). The third condition asserts that the introduction of weak references does not allow the garbage collection rule to collect additional bindings. Proving this condition holds is straightforward. The only non-standard condition is 4, which deserves closer examination.

Operationally, condition 4 means that if we are given a garbage collection rule (**myGarb**) that we would like to use in conjunction with the other rules of  $\lambda_{\text{weak}}$ , then before applying (**myGarb**) to program  $\text{letrec } H^s \uplus H^w \text{ in } e$  we first calculate:

$$\langle H_1, \text{Dom}(H^w), \emptyset \rangle \Rightarrow_{\text{nf}} \langle H', \emptyset, H'' \rangle.$$

After this we can apply (**myGarb**) as long as we only collect bindings from  $H'$ .

We have proven that if the conditions of Definition 5.7 are met then an additional garbage collection rule and the existing rewrite rules of  $\lambda_{\text{weak}}$  can co-exist without violating the semantics of  $\lambda_{\text{weak}}$ . We have also shown how to operationally enforce these conditions.

## 6 Adding Types

We introduce a polymorphic type system for  $\lambda_{\text{weak}}$ . The purpose of this type system is to provide a mechanism to judge whether a program is stuck in the sense of Lemma 2.3. In other words, there only exists valid type derivations for non-stuck programs.

Below we define the types of  $\lambda_{\text{weak}}$ .

### Types:

$$\begin{array}{llll} \text{(type variables)} & \alpha \in \text{TVar} & ::= & \mathbf{a}_1 \mid \mathbf{a}_2 \mid \mathbf{a}_3 \mid \dots \\ \text{(types)} & \tau \in \text{Type} & ::= & \alpha \mid \text{int} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \tau \text{ weak} \\ \text{(type schemes)} & \sigma \in \text{Scheme} & ::= & \tau \mid \forall \alpha. \sigma \end{array}$$

Now we define the typing rules of  $\lambda_{\text{weak}}$ . In the following we use  $\text{FTV}(\Gamma)$  to denote the free type variables of typing context  $\Gamma$ .

### Typing Rules:

#### Expressions:

$$\begin{array}{c}
\frac{}{\Gamma \uplus \{x : \sigma\} \vdash x : \sigma} \text{ (Var)} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \text{ (Pair)} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i e : \tau_i} \text{ (Proj)} \quad (i \in \{1, 2\}) \\
\\
\frac{}{\Gamma \vdash i : \text{int}} \text{ (Int)} \quad \frac{\Gamma \uplus \{x : \tau_1\} \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \text{ (Fun)} \quad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \text{ (App)} \\
\\
\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma \uplus \{x : \sigma\} \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \text{ (Let)} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{weak } e : \tau \text{ weak}} \text{ (Weak)} \quad \frac{}{\Gamma \vdash \mathbf{d} : \tau \text{ weak}} \text{ (D)} \\
\\
\frac{\Gamma \vdash e_1 : \tau' \text{ weak} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau' \rightarrow \tau}{\Gamma \vdash \text{ifdead } e_1 e_2 e_3 : \tau} \text{ (Ifdead)} \\
\\
\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha. \sigma} \text{ (Gen)} \quad (\alpha \notin \text{FTV}(\Gamma)) \quad \frac{\Gamma \vdash e : \forall \alpha. \sigma}{\Gamma \vdash e : \sigma[\alpha \mapsto \tau]} \text{ (Inst)}
\end{array}$$

#### Heaps and Programs:

$$\frac{\forall x \in \text{Dom}(H). \Gamma \uplus \Gamma' \vdash H(x) : \Gamma'(x)}{\Gamma \vdash H : \Gamma'} \text{ (Heap)} \quad \frac{\emptyset \vdash H : \Gamma \quad \Gamma \vdash e : \tau}{\vdash \text{letrec } H \text{ in } e : \tau} \text{ (Prog)}$$

The reader may be curious to know why we have assigned  $\mathbf{d}$  the type:  $\tau \text{ weak}$ . In order to show that the above type system is sound we must show that the type of a program is preserved by the rewrite rules of  $\lambda_{\text{weak}}$ . However, suppose we instead assign  $\mathbf{d}$  the type  $\text{dead}$  and consider the following program evaluation:

$$\begin{array}{l}
\text{letrec } \{\} \text{ in } \langle (\lambda p. \text{weak } p) \langle 5, 6 \rangle, 3 \rangle \\
\begin{array}{l} \xrightarrow{\text{alloc}} \\ \xrightarrow{\text{alloc}} \\ \xrightarrow{\text{alloc}} \\ \xrightarrow{\text{alloc}} \\ \xrightarrow{\text{app}} \\ \xrightarrow{\text{alloc}} \end{array}
\end{array}$$

At this stage in the evaluation the program has type:  $((\text{int} \times \text{int}) \text{ weak}) \times \text{int}$ . But, after the application of one more rewrite rule we have the following:

$$\begin{array}{l}
\text{letrec } \{a \mapsto \lambda p. \text{weak } p, b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c \rangle, f \mapsto \text{weak } e\} \text{ in } \langle f, 3 \rangle \\
\xrightarrow{\text{garb}} \text{letrec } \{f \mapsto \mathbf{d}\} \text{ in } \langle f, 3 \rangle.
\end{array}$$

Now the program has type:  $\text{dead} \times \text{int}$ . Therefore, the type is not strictly preserved by the rewrite relations of  $\lambda_{\text{weak}}$ .

Also consider the following motivation for this typing rule. Suppose we have the following program:

$$\text{letrec } \{a \mapsto 5, x \mapsto \text{weak } a, y \mapsto \mathbf{d}, z \mapsto \lambda b. \text{weak } b\} \text{ in ifdead } x y z$$

Notice that, by typing rule `Ifdead`, we must equate the type of  $\mathbf{d}$  with  $\text{int weak}$ .

By allowing  $\mathbf{d}$  to be assigned type  $\tau \text{ weak}$  for any  $\tau$  whatsoever, we avoid these issues.

Now we show that this type system is sound.

**Theorem 6.1 (Progress).** *For every  $\lambda_{\text{weak}}$  program  $P$ , if  $\vdash P : \tau$  then either  $P$  is an answer or there exists  $P'$  such that  $P \xrightarrow{R} P'$ .  $\square$*

**Theorem 6.2 (Preservation).** *For every  $\lambda_{\text{weak}}$  program  $P$ , if  $\vdash P : \tau$  and  $P \xrightarrow{R} P'$  then  $\vdash P' : \tau$ .  $\square$*

The proofs of these two theorems and all auxiliary lemmas can be found in the appendix. The previous two theorems imply the soundness of the type system for  $\lambda_{\text{weak}}$ . More precisely, they assert that any well-typed  $\lambda_{\text{weak}}$  program is either an answer or it can be reduced one step to a program of the same type.

## 7 Type Inference

We describe a decidable algorithm for inferring the types of the previous section.

### 7.1 Unification

**Definition 7.1 (Substitution).** A substitution,  $S$ , is a total function that maps type variables to types in which only a finite number of type variables are not mapped to themselves. We use the following notation:

1. We write  $S[\alpha \mapsto \tau]$  to mean  $(S - \{(\alpha \mapsto S(\alpha))\}) \cup \{(\alpha \mapsto \tau)\}$ .
2. We write  $S_1; S_2$  for the composition of substitutions  $S_1$  and  $S_2$ . So  $S_1; S_2(\tau)$  is equivalent to  $S_1(S_2(\tau))$ .
3. We write  $S_{\text{Id}}$  to mean  $\{(\alpha \mapsto \alpha) \mid \alpha \in \text{TVar}\}$ .

We apply substitutions inductively on types, and component-wise on typing contexts and constraint sets.  $\square$

**Constraint Unification:**

$$\langle \Delta \cup \{\tau_1 \doteq \tau_2\}, S \rangle \xrightarrow{\text{unify}} \begin{cases} \langle \Delta \cup \{\tau_2' \doteq \tau_1'\} \cup \{\tau_1'' \doteq \tau_2''\}, S \rangle & \text{if } \tau_i = \tau_i' \rightarrow \tau_i'' \text{ and } i \in \{1, 2\} \\ \langle \Delta \cup \{\tau_1' \doteq \tau_2'\} \cup \{\tau_1'' \doteq \tau_2''\}, S \rangle & \text{if } \tau_i = \tau_i' \times \tau_i'' \text{ and } i \in \{1, 2\} \\ \langle \Delta \cup \{\tau_1' \doteq \tau_2'\}, S \rangle & \text{if } \tau_i = \tau_i' \text{ weak and } i \in \{1, 2\} \\ \langle (\alpha \mapsto \tau_2)(\Delta), S[\alpha \mapsto \tau_2] \rangle & \text{if } \tau_1 = \alpha \text{ and } \alpha \notin \text{FTV}(\tau_2) \\ \langle (\alpha \mapsto \tau_1)(\Delta), S[\alpha \mapsto \tau_1] \rangle & \text{if } \tau_2 = \alpha \text{ and } \alpha \notin \text{FTV}(\tau_1) \\ \langle \Delta, S \rangle & \text{if } \tau_1 = \tau_2 \end{cases}$$

### 7.2 Type Inference

**Convention 7.2.**

1. We write  $\text{Gen}(\Gamma, \tau)$  to mean  $\forall \alpha_1 \dots \forall \alpha_n. \tau$  where  $\{\alpha_1 \dots \alpha_n\} = \text{FTV}(\tau) - \text{FTV}(\Gamma)$ .
2. We write  $\Gamma(H)$  to mean  $\{x : \alpha \mid x \in \text{Dom}(H) \text{ and } \alpha \text{ is fresh}\}$ .  $\square$

**Type Inference:**

Below we define the type inference algorithm  $\text{Infer} : \text{Prog} \rightarrow \text{Types}$ . However, first we show two auxiliary algorithms:  $\text{InferExp}$  and  $\text{InferHeap}$ .

**Definition 7.3 (InferExp).**

$$\begin{aligned}
\text{InferExp}(\Gamma, i) &= (S_{\text{Id}}, \text{int}) \\
\text{InferExp}(\Gamma, d) &= (S_{\text{Id}}, \alpha \text{ weak}) \\
&\quad \text{where } \alpha \text{ is fresh} \\
\text{InferExp}(\Gamma, x) &= (S_{\text{Id}}, (\vec{\alpha} \mapsto \vec{\beta})(\tau)) \\
&\quad \text{where } \Gamma(x) = \forall \vec{\alpha}. \tau \text{ and } \vec{\beta} \text{ are fresh} \\
\text{InferExp}(\Gamma, \langle e_1, e_2 \rangle) &= (S_2; S_1, S_2(\tau_1) \times \tau_2) \\
&\quad \text{where } \text{InferExp}(\Gamma, e_1) = (S_1, \tau_1) \text{ and} \\
&\quad \text{InferExp}(S_1(\Gamma), e_2) = (S_2, \tau_2) \\
\text{InferExp}(\Gamma, \pi_i e) &= (S'; S, S'(\alpha_i)) \\
&\quad \text{where } \text{InferExp}(\Gamma, e) = (S, \tau) \text{ and} \\
&\quad \langle \{\tau \doteq \alpha_1 \times \alpha_2\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S' \rangle \text{ and} \\
&\quad \alpha_1, \alpha_2 \text{ are fresh} \\
\text{InferExp}(\Gamma, \lambda x. e) &= (S, S(\alpha) \rightarrow \tau) \\
&\quad \text{where } \text{InferExp}(\Gamma \cup \{x : \alpha\}, e) = (S, \tau) \text{ and} \\
&\quad \alpha \text{ is fresh} \\
\text{InferExp}(\Gamma, e_1 e_2) &= (S'; S_2; S_1, S'(\alpha)) \\
&\quad \text{where } \text{InferExp}(\Gamma, e_1) = (S_1, \tau_1) \text{ and} \\
&\quad \text{InferExp}(S_1(\Gamma), e_2) = (S_2, \tau_2) \text{ and} \\
&\quad \langle \{S_2(\tau_1) \doteq \tau_2 \rightarrow \alpha\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S' \rangle \text{ and} \\
&\quad \alpha \text{ is fresh} \\
\text{InferExp}(\Gamma, \text{let } x = e_1 \text{ in } e_2) &= (S_2; S_1, \tau_2) \\
&\quad \text{where } \text{InferExp}(\Gamma, e_1) = (S_1, \tau_1) \text{ and} \\
&\quad \text{InferExp}(S_1(\Gamma) \cup \{x : \text{Gen}(S_1(\Gamma), \tau_1)\}, e_2) = (S_2, \tau_2) \\
\text{InferExp}(\Gamma, \text{weak } e) &= (S, \tau \text{ weak}) \\
&\quad \text{where } \text{InferExp}(\Gamma, e) = (S, \tau) \\
\text{InferExp}(\Gamma, \text{ifdead } e_1 e_2 e_3) &= (S'_2; S_3; S_2; S'_1; S_1, S'_2; S_3(\tau_2)) \\
&\quad \text{where } \text{InferExp}(\Gamma, e_1) = (S_1, \tau_1) \text{ and} \\
&\quad \langle \{\tau_1 \doteq \alpha \text{ weak}\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S'_1 \rangle \text{ and} \\
&\quad \text{InferExp}(S'_1; S_1(\Gamma), e_2) = (S_2, \tau_2) \text{ and} \\
&\quad \text{InferExp}(S_2; S'_1; S_1(\Gamma), e_3) = (S_3, \tau_3) \text{ and} \\
&\quad \langle \{\tau_3 \doteq S_3(S_2; S'_1(\alpha) \rightarrow \tau_2)\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S'_2 \rangle \text{ and} \\
&\quad \alpha \text{ is fresh}
\end{aligned}$$

**Definition 7.4 (InferHeap).**

$$\text{InferHeap}(\Gamma, H) = \begin{cases} \Gamma & \text{if } H = \emptyset \\ \text{InferHeap}(S'; S(\Gamma), H') & \text{if } H = H' \uplus \{x \mapsto hv\} \text{ and} \\ & \text{InferExp}(\Gamma, hv) = (S, \tau) \text{ and} \\ & \langle \{S(\Gamma(x)) \doteq \tau\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S' \rangle \end{cases}$$

**Definition 7.5 (Infer).**

$$\text{Infer}(\text{letrec } H \text{ in } e) = \tau \quad \text{if } \text{InferHeap}(\Gamma(H), H) = \Gamma \text{ and} \\
\text{InferExp}(\Gamma, e) = (S', \tau)$$

Note: If any of the conditions of `Infer` are not met then `Infer` fails.

REMARK 7.6. Note that the preceding inference algorithm is a variant of the well-known Algorithm W [DM82]. □

**Theorem 7.7 (Soundness of Infer).** *For any  $\lambda_{weak}$  program  $P$ , if  $\text{Infer}(P) = \tau$  then  $\vdash P : \tau$ .* □

**Theorem 7.8 (Completeness and Principality of Infer).** *For any  $\lambda_{weak}$  program  $P$ , if  $\vdash P : \tau$  then  $\text{Infer}(P) = \tau'$  and there exists a substitution  $S$  such that  $S(\tau') = \tau$ .* □

The proofs of these theorems and all auxiliary lemmas can be found in the appendix.

## 8 Summary

This work provides a framework for formally reasoning about the semantics of weak references, a troublesome programming feature. We borrow from  $\lambda_{gc}$  as defined in [MFH95] the idea of utilizing  $\lambda$ -calculus techniques to reason about memory management by making the heap and memory allocation syntactically explicit. This allows us to model the conventional behavior of weak references in a compact manner that is amenable to rigorous proof.

The  $\lambda_{weak}$  framework is also flexible, as is shown by our ability to easily restrict the calculus in sections 3.1, 3.2 and 4, and extend it in section 3.3. We expect further variations on the semantics of weak references to be defined within the  $\lambda_{weak}$  framework without much difficulty.

## References

- [Car] Carnegie Mellon University, <http://common-lisp.net/project/cmucldoc/cmucldoc/user/extensions.html#toc34>. *CMUCL User's Manual: Design Choices and Extensions*.
- [CMP00] E. Chailloux, P. Manoury, and B. Pagano. *Developing Applications with Objective Caml*. O'Reilly, France, 2000.
- [DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Conf. Rec. 9th Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 207–212, 1982.
- [GNU] GNU, <http://clisp.sourceforge.net/impnotes.html>. *Implementation Notes for GNU CLISP*.
- [Gui] The Guile Scheme Team, [http://www.fokus.gmd.de/gnu/docs/guile/guile\\_46.html](http://www.fokus.gmd.de/gnu/docs/guile/guile_46.html). *Guile Scheme - Weak References*.
- [Hug98] The Hugs-GHC Team, <http://www.dcs.gla.ac.uk/fp/software/ghc/lib/hg-libs-15.html>. *The Hugs-GHS Extension Libraries: Weak*, Aug. 1998.
- [JME99] S. P. Jones, S. Marlow, and C. Elliott. Stretching the storage manager: weak pointers and stable names in haskell. In *Proceedings of the Workshop on Implementing Functional Languages*, 1999.
- [MFH95] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Conference on Functional Programming Languages and Computer Architecture*, pp. 66–77, June 1995.

- [Mosa] The Moscow ML Team, <http://www.dina.dk/~sestoft/mosmlib/Weak.html>. *Moscow ML Library Manual: Structure Weak*.
- [Mosb] The Moscow ML Team, <http://www.dina.dk/~sestoft/manual/node9.html#SECTION00090000000000000000>. *Moscow ML Owner's Manual: Weak Pointers*.
- [MZC04] F. Merizen, O. Zendra, and D. Colnet. Designing efficient and safe non-strong references in eiffel with parametric types. Technical Report LORIA A04-R-149, INRIA-Lorraine, France, Sept. 2004.
- [OCa] The OCaml Team, <http://caml.inria.fr/ocaml/htmlman/libref/Weak.html>. *OCaml 3.04: Module Weak*.
- [PLT] The PLT Scheme Team, [http://download.plt-scheme.org/doc/mzscheme/mzscheme-Z-H-13.html#node\\_sec\\_13.1](http://download.plt-scheme.org/doc/mzscheme/mzscheme-Z-H-13.html#node_sec_13.1). *PLT MzScheme: Language Manual*.
- [Pyt] The Python Team, <http://docs.python.org/lib/module-weakref.html>. *Python Library Reference: 3.3 weakref - Weak references*.
- [SML] The SMLofNJ Team, <http://www.smlnj.org/doc/SMLofNJ/pages/weak.html>. *The SMLofNJ Structure: The Weak Signature*.
- [Sun] Sun Microsystems, <http://java.sun.com/j2se/1.3/docs/api/java/lang/ref/package-summary.html>. *Java 2 Platform SE v1.3.1: Class WeakReference*.
- [TT97] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997.

## A A Weak Hash-Consing Example

In this appendix we encode an example use of weak references that closely resembles their use in practice and make use of our semantics by proving a space consumption result. In particular, we encode an implementation of hash-consing with weak references in  $\lambda_{\text{weak}}$ . For easy of reading we extend  $\lambda_{\text{weak}}$  with lists and conditionals (note that both can be encoded by the syntax of  $\lambda_{\text{weak}}$  so they may be considered syntactic sugar). We use the conventional semantics for these programming features and do not define them explicitly here. Lastly, we assume we have a function `equals` to test the equality of two lists.

The example is as follows.

```

letrec {}
in let w0 = weak []
    in let w1 = weak [1]
        in let table = ⟨w0, w1⟩
            in let hash = λ.
                if equals (l, [])
                then 0
                else if equals (l, [1])
                then 1
                else - 1

```

```

in let hashCons = λw0.λw1.λl.
  if (hash l) = 0
  then ifdead w0 ((λl.⟨l, ⟨weak l, w1⟩⟩) []) (λl.⟨l, ⟨weak l, w1⟩⟩)
  else if (hash l) = 1
       then ifdead w1 ((λl.⟨l, ⟨w0, weak l⟩⟩) [1]) (λl.⟨l, ⟨w0, weak l⟩⟩)
       else ⟨[-1], ⟨w0, w1⟩⟩
in let p = hashCons w0 w1 []
    in let p' = hashCons (π1 π2 p) (π2 π2 p) []
        in ⟨π1 p, π1 p'⟩

```

We now show the utility of this calculus by sketching a proof of a result about the space consumption of this program. First we show that this program correctly hash-cons lists. Note that in the following we use the term sublist of  $l$  to refer any  $l'$  such that there exists a  $l''$  that when prepended to  $l'$  equals  $l$ .

**Lemma A.1 (Correctness of Hash-Consing).** *Let  $P$  be a  $\lambda_{\text{weak}}$  program that contains the definition of  $\text{hashCons}$ , where  $\text{table}$  is not manipulated otherwise, and:*

$$P \xrightarrow{R,*} \text{letrec } H_1 \text{ in } E[\text{hashCons } l] \xrightarrow{R,*} \text{letrec } H_2 \text{ in } E[x] \xrightarrow{R,*} \text{letrec } H_3 \text{ in } e$$

where  $l$  is a non-[-1]-list. If  $x \in \text{Dom}(H_3^s)$  then for all sublists  $l'$  of  $l$  there is a unique representation of  $l'$  in  $\text{table}$  in  $H_3$ , i.e. from the location of  $\text{table}$  in  $H_3$  there is a unique path of pointers to a representation of  $l'$ .  $\square$

*Proof Sketch.*

- By investigating the definition of  $\text{hashCons}$  and the rewrite rules of  $\lambda_{\text{weak}}$ , and the fact that  $\text{table}$  is only manipulated by  $\text{hashCons}$  we know the following: if  $P \xrightarrow{R,*} \text{letrec } H \text{ in } e$ , then  $\text{table}$  contains no duplicate representations of lists in  $H$ .
- By induction on the size of  $l$  we show: there is a unique representation of every sublist of  $l$  in  $\text{table}$  in  $H_2$ .

*Base Case:*  $l = []$

Then applying  $\text{hashCons}$  to  $l$  returns  $e$  which is a unique representation of  $[]$  in  $\text{table}$  in  $H_2$ . Note,  $e$  can't be garbage collected since  $e$  is reachable from  $\text{hashCons}$ .

*Inductive Case:*  $l = \text{cons } y \text{ } ys$

Then by the induction hypothesis all sublists of  $ys$  have a unique representation in  $\text{table}$  in  $H_2$ . In particular  $ys$  does. If  $\text{lookUp } l \text{ } (! \text{table})$  does not return  $[-1]$  then  $l$  is in  $\text{table}$  and therefore its representation is unique. Otherwise  $\text{insert } (\text{cons } y \text{ } (\text{lookUp } ys \text{ } (! \text{table})))$  is returned because  $\text{lookUp } ys \text{ } (! \text{table})$  does not return  $[-1]$ . In this case,  $\text{cons } y \text{ } ys$  is added to  $\text{table}$  and it is a unique representation. Note  $\text{cons } y \text{ } ys$  cannot be garbage collected from  $H_2$  because it is reachable via  $x$ .

- By investigation of the rewrite rules and the definition of  $\text{hashCons}$  we know  $H_2(x)$  and  $H_3(x)$  are representations of  $l$ .
- Therefore the unique representation of  $l'$  in  $\text{table}$  in  $H_2$  must still exist in  $\text{table}$  in  $H_3$  since  $x \in \text{Dom}(H_3^s)$ .  $\square$

Now we prove that optimal garbage collection implies that each list in *table* is reachable from outside *table*.

**Lemma A.2.** *Let  $P$  be a  $\lambda_{\text{weak}}$  program that contains the definition of `hashCons`, where *table* is not manipulated otherwise, and:*

$$P \xrightarrow{R,*} P' \xrightarrow{\text{garb}} \text{letrec } H \text{ in } e.$$

*Then *table* contains list  $l$  in  $H$  only if there exists  $x \in \text{Dom}(H^s)$  such that  $H(x)$  is a representation of  $l$ .*  $\square$

*Proof Sketch.* Let  $H(y)$  be the representation of  $l$  contained by *table* in  $H$ . In other words, there is a path of pointers from the location of *table* in  $H$  to some binding  $(w \mapsto \text{weak } y) \in H$ .

Now suppose for the sake of contradiction that there is no  $y \in \text{Dom}(H^s)$  such that  $H(y)$  is a representation of  $l$ . Then by the definition of `(garb)` the binding  $(w \mapsto \text{weak } y)$  would be removed from  $H$  and  $(w \mapsto \text{weak } y) \notin H$ .  $\square$

Lastly we combine the previous two lemmas to show that optimal garbage collection implies that our implementation of hash-consing uses optimal space. We define optimal space for our implementation of hash-consing as follows.

**Definition A.3 (Optimal Space for Hash-Consing).** A  $\lambda_{\text{weak}}$  program  $P$  that implements hash-consing consumes optimal space if the following holds,

1. the hash table of  $P$  doesn't contain any garbage collectible lists, that is, every list in the hash table of  $P$  is reachable according to `(garb)` from  $P$
2. the hash table of  $P$  doesn't contain any duplicate lists.  $\square$

**Theorem A.4 (*hashCons* Uses Optimal Space).** *Assuming optimal garbage collection, in the sense of Lemma A.2, our implementation of hash-consing consumes an optimal amount of space, in the sense of Definition A.3.*  $\square$

*Proof Sketch.* Lemma A.2 tells us a list is in *table* only if it is reachable outside the hash-consing implementation. So every list in *table* is needed. Lemma A.1 says each sublist reachable outside the hash-consing implementation has a unique representation in *table*. So *table* doesn't contain any duplicate entries.  $\square$

## B A Survey of Weak References in Popular Programming Languages

This appendix surveys the semantics of weak references in many popular programming languages. Of course, all the languages that are considered must provide automatic memory management (otherwise discussing weak references becomes moot) so several well-known programming languages are immediately disqualified from consideration. The language implementations considered are: ML (SMLofNJ [SML], MoSML [Mosa], OCAML [OCa]), Haskell (GHC [Hug98]), dialects of Lisp (Common Lisp: CMUCL [Car], GNU CLISP [GNU] and Scheme: Guile Scheme [Gui], PLT MzScheme [PLT]), Python [Pyt], Java [Sun], and Eiffel [MZC04].

## B.1 Simple Semantics of Weak References (SMLofNJ, CMUCL, PLT MzScheme)

The semantics of weak references as implemented in SMLofNJ are, in our view, the most basic. These can be best explained by the following type signature:

```
type 'a weak
val weak : 'a -> 'a weak
val strong : 'a weak -> 'a option
```

'a **weak** is the type of a weak pointer to an object of type 'a. The function **weak** constructs a weak pointer to whatever it is applied to. The function **strong**, when applied to a weak pointer, dereferences the pointer, returning **NONE** if the object pointed to has been garbage collected and **SOME(a)** if the pointer references the still-alive object **a**.

In addition, SMLofNJ provides opaque weak references. These are weak references that can be examined to determine if the referenced object has been garbage collected, but the underlying object cannot be extracted from them. The purpose of these weak references is “to make finalizers (of heterogenous collections) statically type-check.” [SML]

The semantics of weak references in CMUCL, an implementation of Common Lisp is slightly different than those of SMLofNJ. This implementation does not provide opaque weak references and the dereferencing function, instead of returning an option type, returns a pair of the object pointed to (or **NIL** if the object has been garbage collected) and a boolean value denoting whether the referenced object has been garbage collected or not.

PLT MzScheme, an implementation of Scheme which is a dialect of Lisp, also defines weak references in a slightly different way. Here weak references are created in the same way as above but when extracting the object pointed to by a weak reference PLT MxScheme returns **#f**, the symbol for false, when the referenced object has been garbage collected. The fact that PLT MzScheme is untyped gives the language enough flexibility to allow for the possibility of these semantics.

## B.2 Weak Collections (MoSML, OCaml, GNU CLISP, Guile Scheme)

Another implementation of SML, MoSML, provides a slight variation on the above semantics of weak references. MoSML provides a dereferencing function

```
val get : 'a weak -> 'a
```

which returns the value pointed to by the weak reference if it has not been garbage collected. However, if the object has been collected then MoSML raises the exception: **Fail "Dangling weak pointer"**. To quote the MoSML documentation: “We raise an exception instead of returning an option value, because access via a weak pointer to a deallocated object is likely to be a programming error [Mosa].”

Another distinguishing difference between weak references in SMLofNJ and MoSML is that MoSML defines *weak arrays*. A weak array is similar to a standard array except that each element is pointed to by a weak reference. Therefore the elements of a weak array can be deallocated by garbage collection if no strong reference is keeping them alive. “Weak vectors are ideal to implement object recycling ...” [MZC04]

The MoSML documentation also notes that a value of type 'a **Weak.weak** (a simple weak references) is equivalent to, but more efficient than, a single-element 'a **Weak.array**. Conversely, an 'a **Weak.array** is more efficient than an ('a **Weak.weak**) **Array.array**.

The semantics of weak references in GNU CLISP is identical to that of CMUCL with the addition of weak hash tables. The idea behind weak hash tables is similar to that of weak arrays, but now we have three varieties of weak hash tables depending on where the weak references occur. A *weak-key hash table* contains key-value pairs as a traditional hash table would. But here, the key is weakly referenced. When the key is garbage collected the entire key-value pair is removed from the hash table. In a *weak-value hash table* the value is weakly referenced and, as before, when the value is garbage collected the entire key-value pair is removed from the hash table. Lastly *doubly-weak hash table* weakly references both the key and the value. If either object is garbage collected then the entire pair is removed from the hash table. Weak hash tables “... can be used to attach meta information to objects without keeping the objects [a]live more than necessary.” [MZC04]

The weak reference semantics of OCAML and Guile Scheme are similar to that of SMLofNJ and PLT MzScheme respectively, but both implementations allow weak arrays and weak hash tables as their only form of weak references.

### B.3 Finalization and Weak References (GHC)

The Glasgow Haskell Compiler (GHC) intimately connects weak references with the process of finalization. That is, when creating a weak reference in GHC, one must supply both the object to be referenced and a finalizer function to be run (exactly once) after the weakly referenced object has been garbage collected. An example finalizer function is one that removes key-value pairs from a hash table once the weakly referenced key has been garbage collected (as described above).

The type signature of the weak reference related functions in GHC is as follows

```
mkWeakPtr :: a -> IO () -> IO (Weak a)
deRefWeak :: Weak a -> IO (Maybe a).
```

As in SMLofNJ, GHC uses the equivalent of an option type when dereferencing weak pointers. In keeping with Haskell’s purity mandate, GHC wraps the resulting weak reference in the IO monad. This because it has the side effect of arranging that the finaliser will be run when the object dies.

GHC also defined a generalized interface for weak references which is provided “... to implement memo tables properly.” [Hug98] The interface is

```
mkWeak :: k -> v -> IO () -> IO (Weak v).
```

`mkWeak` takes a key of any type `k`, a value of any type `v`, and a finalizer and returns a weak reference to the value. As defined above, `deRefWeak` returns the value only, not the key. The purpose of this generalized interface is to prevent problems like the following:

Suppose there is a memo table in which the value of a key-value pair contains a reference to the key. Now, the memo table keeps the value alive, which keeps the key alive, even though there may be no other references to the key so both should die.

In this generalized interface, `deRefWeak` returns `Nothing` if the key, rather than the value, has been garbage collected. Also, references from the value to the key do not keep the key alive, just as the finalizer does not keep the key alive.

Lastly we note that the original presentation of GHC’s weak references can be defined in terms of these more general weak references:

```
mkWeakPtr :: a -> IO () -> IO (Weak a)
mkWeakPtr v f = mkWeak v v f.
```

## B.4 Proxies and Weak References (Python)

All the semantics of weak references described above require weak pointers to be dereferenced in order to manipulate the underlying object. The implementation of weak references in Python tries to abstract away this level of indirection. Python uses *proxy objects*, which weakly reference an object while at the same time disguise themselves as the object. So, the interface of the proxy object (the weak reference) is the same as the interface of the underlying object itself. All operations on the proxy are relayed to the underlying object if it has not been garbage collected, otherwise an exception is thrown.

Although the functionality of a proxy object is equivalent to that of the referenced object, programming with a proxy is much less efficient since the proxy must make sure the underlying object has not been garbage collected before each access. Also, programming with proxy objects can be very difficult since proxies can be used anywhere in a program that the underlying object can. This means that exceptions thrown by a proxy in a program can be challenging to track down. As a result, debugging and proving correctness results can be very difficult. [Pyt]

## B.5 Advanced Weak Reference (Java, Eiffel)

Since its 1.2 specification, the Java language has included a form of weak references whose semantics are essentially equivalent to those of SMLofNJ. However, Java has also implemented extended forms of weak references with varying strengths (listed by decreasing strength):

*Soft references* (`java.lang.ref.SoftReference` class): Soft references are references to objects which are cleared from memory at the discretion of the garbage collector (based on its space demands). Informally, the garbage collector is designed to “try to keep” softly referenced objects alive “as long as possible” after the object is no longer strongly reachable. However, there are no guarantees made about the time when the garbage collector will deallocate a softly referenced object. According to the Java specification, “Soft references are for implementing memory-sensitive caches.” [Sun]

*Weak references* (`java.lang.ref.WeakReference` class): Weak references, as mentioned above, have essentially the same semantics as those of SMLofNJ. If an object is only weakly reachable then the garbage collector will deallocate it (unlike soft references there is no attempt made to delay the deallocation). “weak references are for implementing canonicalizing mappings that do not prevent their keys (or values) from being reclaimed.” [Sun]

*Phantom references* (`java.lang.ref.PhantomReference` class): Phantom references are operationally equivalent to the opaque references of SMLofNJ. These are finalization tools which cannot be dereferenced (to ensure that reclaimable objects remain so). “Phantom references are most often used for scheduling pre-mortem cleanup actions in a more flexible way than is possible with the Java finalization mechanism.” [Sun] Once the garbage collector determines that an object referenced by a phantom reference is reachable only by phantom references, the garbage collector will enqueue the phantom reference. A `ReferenceQueue` allows a program to be notified of changes to an object’s reachability. After a reference is enqueued, the program may remove references from the queue by either polling or blocking until a reference becomes available. Although phantom references are enqueued by default, this operation is optional for soft and weak references.

The implementors of SmartEiffel, the GNU Eiffel compiler, are currently attempting to further extend

the programmers control over memory management via weak references [MZC04]. The proposal will extend SmartEiffel with soft references as described for Java above. Recall that whether or not a softly referenced object is deallocated depends on the heuristics built into the design of the garbage collector. However, the implementors of SmartEiffel note that no single heuristic will be suitable for all situations. Therefore, a goal is to allow the programmer to define one’s own heuristics.

*Tunable references:* The first attempt by SmartEiffel to achieve this goal will be the implementation of tunable references. The idea behind tunable references is to allow the programmer to choose the “strength” of a weak references. This allows the programmer to define an ordering of soft reference deallocation that the garbage collector will respect.

*Programmable references:* The second, and more drastic, attempt by SmartEiffel to give the programmer more control over the heuristic for soft reference deallocation is called programmable references. Programmable references are an attempt to let the programmer write their own heuristics. When a programmable reference is created the developer must define a `item_reclamation_allowed` function which will be called by the garbage collector when the referenced object is available for deallocation, determining whether or not the object is acutally removed. This approach does have the added benefit of lightening the burden of the garbage collector designer.

## C Proof of GC Oblivious Properties

### C.1 Proof of Polynomial-Time Algorithm for Membership in $\text{Exp}^*$

**Proposition C.1 (Efficient Test for Membership in  $\text{Exp}^*$  and  $\text{ExpPair}(p)$ ).** *Let  $e_1, e_2$  be arbitrary expressions. There are effective procedures to decide whether  $e_1$  is in the restricted set  $\text{Exp}^*$  and whether  $(e_1, e_2)$  is in  $\text{ExpPair}(p)$  for all  $p \in \{1, 2\}^*$ . The procedures run in time which is a (low-degree) polynomial in the sizes of  $e_1, e_2$ .  $\square$*

*Proof.* This proof proceeds by three nestings of induction. The proof begins by induction on the number of occurences of  $\pi_i$  in  $e_1$ .

*Base Case:* There are no occurences of  $\pi_i$  in  $e_1$ .

We prove this by induction on the length of  $p$ .

*Base Case:*  $p = \varepsilon$ .

We prove this by induction on the structure of  $e_1$ .

*Case:* If  $e_1$  is  $x$  or  $i$  then it is easy to decide that  $e_1 \in \text{Exp}^*$  and  $(e_1, e_2) \notin \text{ExpPair}(\varepsilon)$ .

*Case:* If  $e_1 = \langle e', e'' \rangle$ , then by the induction hypothesis on  $e_1$  we have an effective procedure to determine whether  $e', e'' \in \text{Exp}^*$ . If both are in  $\text{Exp}^*$  then  $e_1 \in \text{Exp}^*$ . It is easy to see that  $(e_1, e_2) \notin \text{ExpPair}(\varepsilon)$ .

*Case:*  $e_1 \neq \pi_i e'$  since there are no occurences of  $\pi_i$  in  $e_1$ .

*Case:* If  $e_1 = \lambda x.e'$ , then by the induction hypothesis on  $e_1$  we have an effective procedure to determine whether  $e' \in \text{Exp}^*$ . If so, then  $e_1 \in \text{Exp}^*$ . To decide  $(e_1, e_2) \in \text{ExpPair}(\varepsilon)$  first we must decide whether  $e_2 = \lambda x.e''$  for some expression  $e''$ . If not then  $(e_1, e_2) \notin \text{ExpPair}(\varepsilon)$ . Otherwise, by the induction hypothesis on  $e_1$  (again) we can effectively decide whether  $(e', e'') \in \text{ExpPair}(\varepsilon)$ . If so, then  $(e_1, e_2) \in \text{ExpPair}(\varepsilon)$ , otherwise,  $(e_1, e_2) \notin \text{ExpPair}(\varepsilon)$ .

*Case:* If  $e_1 = e'_1 e_3$ , then by the induction hypothesis on  $e_1$  we have an effective procedure to determine whether  $e'_1, e''_1 \in \text{Exp}^*$ . If both are in  $\text{Exp}^*$  then  $e_1 \in \text{Exp}^*$ . To decide  $(e_1, e_2) \in \text{ExpPair}(\varepsilon)$  first we must decide whether  $e_2 = e'_2 e_3$  for some expression  $e'_2$ . If not then  $(e_1, e_2) \notin \text{ExpPair}(\varepsilon)$ . Otherwise, by the induction hypothesis on  $e_1$  (again) we can effectively decide whether  $(e'_1, e'_2) \in \text{ExpPair}(\varepsilon)$ . If so, then  $(e_1, e_2) \in \text{ExpPair}(\varepsilon)$ , otherwise,  $(e_1, e_2) \notin \text{ExpPair}(\varepsilon)$ .

*Case:* If  $e_1 = (\text{let } x = e' \text{ in } e'')$  then  $e_1 \notin \text{Exp}^*$  and  $(e_1, e_2) \notin \text{ExpPair}(\varepsilon)$ .

*Case:* If  $e_1 = \text{weak } e'$ , then by the induction hypothesis on  $e_1$  we have an effective procedure to determine whether  $e' \in \text{Exp}^*$ . If so, then  $e_1 \in \text{Exp}^*$ . To decide  $(e_1, e_2) \in \text{ExpPair}(\varepsilon)$  first we must decide whether  $e_2 = e'$ . If not then  $(e_1, e_2) \notin \text{ExpPair}(\varepsilon)$ . Otherwise,  $e' \in \text{Exp}^*$  determines whether  $(e_1, e_2) \in \text{ExpPair}(\varepsilon)$ .

*Case:* If  $e_1 = (\text{ifdead } e'_1 (e'''_1 e''_1) e'''_1)$ , then by the induction hypothesis on  $e_1$  we have effective procedures to determine whether  $e'''_1 \in \text{Exp}^*$  and whether  $(e'_1, e''_1) \in \text{ExpPair}(\varepsilon)$ . If both tests succeed then  $e_1 \in \text{Exp}^*$ . To decide  $(e_1, e_2) \in \text{ExpPair}(\varepsilon)$  first we must decide whether  $e_2 = (\text{ifdead } e'_1 (e'_2 e''_1) e'_2)$  for some expression  $e'_2$ . If not then  $(e_1, e_2) \notin \text{ExpPair}(\varepsilon)$ . Otherwise,  $(e'_1, e''_1) \in \text{ExpPair}(\varepsilon)$  and  $(e'''_1, e'_2) \in \text{ExpPair}(\varepsilon)$  determines whether  $(e_1, e_2) \in \text{ExpPair}(\varepsilon)$ . Both of these tests are effectively decidable by the induction hypothesis on  $e_1$ .

*Inductive Case:*  $p = i_k \cdots i_2 i_1$ .

We prove this by induction on the structure of  $e_1$ . All of the case are the same as those listed in the base case (above) with the exception of the following.

*Case:* If  $e_1 = \langle e'_1, e''_1 \rangle$ , then by the induction hypothesis on  $e_1$  we have an effective procedure to determine whether  $e'_1, e''_1 \in \text{Exp}^*$ . If both are in  $\text{Exp}^*$  then  $e_1 \in \text{Exp}^*$ . There are two subcases needed to determine whether  $(e_1, e_2) \in \text{ExpPair}(\varepsilon)$ .

*Subcase:*  $i_k = 1$ . We must first decide whether  $e_2 = \langle e'_2, e''_1 \rangle$  for some expression  $e'_2$ . If so then  $(e_1, e_2) \in \text{ExpPair}(\varepsilon)$  holds if  $(e'_1, e'_2) \in \text{ExpPair}(i_{k-1} \cdots i_2 i_1)$  and  $e''_1 \in \text{Exp}^*$ . The first condition follows from the induction hypothesis on the length of  $p$  and the second condition from the induction hypothesis on  $e_1$ .

*Subcase:*  $i_k = 2$ . Similar to the previous subcase.

*Inductive Case:* There are  $n$  occurrences of  $\pi_i$  in  $e_1$ .

We prove this by induction on the length of  $p$ . This proof is the same as for the base case (above) except both the base case and the inductive case of the proof by induction on the length of  $p$  must contain the following case.

*Case:* If  $e_1 = \pi_i e'$  then by the induction hypothesis on  $e_1$  we have an effective procedure to determine whether  $e' \in \text{Exp}^*$ . If so, then  $e_1 \in \text{Exp}^*$ . To decide whether  $(e_1, e_2) \in \text{ExpPair}(p)$  we must first determine whether  $e_2 = \pi_i e''$  for some expression  $e''$ . If not then  $(e_1, e_2) \notin \text{ExpPair}(p)$ . Otherwise  $(e_1, e_2) \in \text{ExpPair}(p)$  if  $(e', e'') \in \text{ExpPair}(ip)$ . By induction on the number of occurrences of  $\pi_i$  in  $e_1$  there is an effective procedure to determine this.  $\square$

## C.2 Proof GC-Oblivious Programs Are Well-Behaved

**Definition C.2.** The following rule allows us to perform garbage collection in a non-deterministic manner.

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

where we denote the reflexive, transitive closure of  $\xrightarrow{\text{gc}}$  and  $\xrightarrow{\text{weak-gc}}$  as  $\xrightarrow{\text{gc},*}$  and  $\xrightarrow{\text{weak-gc},*}$ , respectively. Notice that (garb') may not collect all the possible garbage bindings of a heap. This is not the case with (garb). In fact, (garb) is a special case of (garb').  $\square$

**Definition C.3.** The following rule allows garbage to be added to the heap of a program.

$$\text{(add)} \quad \text{letrec } H \text{ in } e \xrightarrow{\text{add}} \text{letrec } H \uplus H' \text{ in } e \\ \text{provided } \text{Dom}(H') \cap \text{FV}(\text{letrec } H \text{ in } e) = \emptyset$$

$\square$

**Lemma C.4.** If  $P_1 \xrightarrow{\{\text{garb}', \text{add}\},*} (\text{letrec } H_1 \text{ in } x_1)$ , then there exists  $(\text{letrec } H_2 \text{ in } x_2)$  such that

1.  $P_1 \xrightarrow{\text{garb}',*} (\text{letrec } H_2 \text{ in } x_2)$ ,
2.  $\text{result}(H_1, x_1) = \text{result}(H_2, x_2)$ , and
3.  $H_2 \subseteq H_1$ .

$\square$

*Proof Sketch.* This is proved by induction on the length of the reduction sequence  $P_1 \xrightarrow{\{\text{garb}', \text{add}\},*} (\text{letrec } H_1 \text{ in } x_1)$ . Since (add) only inserts garbage into the heap, it is not difficult to see that it cannot affect the result of the program.  $\square$

**Lemma C.5.** If  $\text{result}(H_1, e) = \text{result}(H_2, e)$ , then  $(\text{letrec } H_1 \text{ in } e) \xrightarrow{\text{garb}'} (\text{letrec } H'_1 \text{ in } e) \xrightarrow{\text{add}} (\text{letrec } H_2 \text{ in } e)$ .

$\square$

*Proof Sketch.* To prove this Lemma we first show that it is possible to apply (garb') such that  $\text{result}(H'_1, e) = \text{result}(H_2, e)$  and  $H'_1 \subseteq H_2$ . This is proved by induction on the size of  $H_1$ , using case analysis on the bindings.

Next we prove that if  $(x \mapsto hv) \in (H_2 - H'_1)$ , then  $x \notin \text{FV}(\text{letrec } H'_1 \text{ in } e)$ . Therefore, the (add) can insert this binding into  $H'_1$ . This is proved by induction on the size of  $H_2 - H'_1$ .  $\square$

**Lemma C.6.** If  $(e_1, e_2) \in \text{ExpPair}$ ,  $H$  is a heap, and

$$\text{letrec } H \text{ in } E[\text{ifdead } e_1 (e_3 e_2) e_3] \xrightarrow{\text{R-}\{\text{garb}\},*} \text{letrec } H' \text{ in } E[\text{ifdead } x (e_3 e_2) e_3],$$

then

$$\text{letrec } H' \text{ in } E[\text{ifdead } x (e_3 e_2) e_3] \xrightarrow{\text{ifdead}} \text{letrec } H' \text{ in } E[e_3 y],$$

where  $H'(x) = \text{weak } y$ .  $\square$

*Proof Sketch.* We can prove by induction on the derivation of  $(e_1, e_2) \in \text{ExpPair}$  that if

$$\text{letrec } H \text{ in } E[\text{ifdead } e_1 (e_3 e_2) e_3] \xrightarrow{\text{R-}\{\text{garb}\}}^* \text{letrec } H' \text{ in } E[\text{ifdead } (\text{weak } y) (e_3 e_2) e_3]$$

then it does so when  $H$  is empty. Therefore  $(\text{weak } y)$  is allocated to the heap during the evaluation of

$$\text{letrec } H \text{ in } E[\text{ifdead } e_1 (e_3 e_2) e_3] \xrightarrow{\text{R-}\{\text{garb}\}}^* \text{letrec } H' \text{ in } E[\text{ifdead } x (e_3 e_2) e_3].$$

Since there is no garbage collection during this evaluation we have  $H'(x) = \text{weak } y$ .  $\square$

**Definition C.7.** Given heaps  $H_1, H_2$  and expression  $e$  we write  $H_1 \approx_e H_2$  to mean for all  $x \in \text{FV}(e)$  either

- $\text{result}(H_1, x) = \text{result}(H_2, x)$  or
- $H_1(x) = \text{d}$  and  $H_2(x) = \text{weak } y$  for some  $y$ .

$\square$

**Lemma C.8.** *Suppose  $H_1, H_2$  are heaps and  $e \in \text{Exp}^*$ . If  $H_1 \approx_e H_2$  and*

$$\text{letrec } H_1 \text{ in } E_1[e] \xrightarrow{\text{R-}\{\text{garb}\}}^* \text{letrec } H'_1 \text{ in } E_1[x],$$

*then there exists a unique reduction sequence*

$$\text{letrec } H_2 \text{ in } E_2[e] \xrightarrow{\text{R-}\{\text{garb}\}}^* \text{letrec } H'_2 \text{ in } E_2[x]$$

*such that  $H'_1 \approx_x H'_2$ .*  $\square$

*Proof.* This proof proceeds by induction on the length of  $(\text{letrec } H_1 \text{ in } E_1[e]) \xrightarrow{\text{R-}\{\text{garb}\}}^* (\text{letrec } H'_1 \text{ in } E_1[x])$ .

*Base case:* If the length is 0, then  $e = x$  and the result follows immediately from the hypothesis.

*Inductive case:* We proceed by case analysis on the first rewrite rule applied.

*case:*  $(\text{letrec } H_1 \text{ in } E_1[e]) = (\text{letrec } H_1 \text{ in } E_1[E[hv]]) \xrightarrow{\text{alloc}} (\text{letrec } H_1 \uplus \{y \mapsto hv\} \text{ in } E_1[E[y]]) \xrightarrow{\text{R-}\{\text{garb}\}}^* (\text{letrec } H'_1 \text{ in } E_1[x])$

Then the unique rewrite rule that applies to  $(\text{letrec } H_2 \text{ in } E_2[e]) = (\text{letrec } H_2 \text{ in } E_2[E[hv]])$  is

$$(\text{letrec } H_2 \text{ in } E_2[e]) = (\text{letrec } H_2 \text{ in } E_2[E[hv]]) \xrightarrow{\text{alloc}} (\text{letrec } H_2 \uplus \{y \mapsto hv\} \text{ in } E_2[E[y]]).$$

By the hypothesis of the Lemma we have  $H_1 \approx_{E[hv]} H_2$ . Therefore we have  $(H_1 \uplus \{y \mapsto hv\}) \approx_{E[y]} (H_2 \uplus \{y \mapsto hv\})$ . Now we can apply the inductive hypothesis to get a unique reduction sequence

$$\text{letrec } H_2 \uplus \{y \mapsto hv\} \text{ in } E_2[E[y]] \xrightarrow{\text{R-}\{\text{garb}\}}^* \text{letrec } H'_2 \text{ in } E_2[x]$$

such that  $H'_1 \approx_x H'_2$ . This finishes the case.

*case:* If the first rule applied is  $(\pi_i)$  or  $(\text{app})$  then the proof proceeds similar to above.

*case:* The first rule applied cannot be  $(\text{let-in})$  because  $e \in \text{Exp}^*$  and  $\text{Exp}^*$  does not allow let-in-expressions.

case:  $(\text{letrec } H_1 \text{ in } E_1[E[\text{ifdead } y (e_3 e_2) e_3]]) \xrightarrow{\text{ifdead}} (\text{letrec } H_1 \text{ in } E_1[E[e_3 w]]) \xrightarrow{R\text{-}\{\text{garb}\}^*} (\text{letrec } H'_1 \text{ in } E_1[x])$

In fact,  $(\text{letrec } H_1 \text{ in } E_1[e]) \neq (\text{letrec } H_1 \text{ in } E_1[E[\text{ifdead } y (e_3 e_2) e_3]])$  since  $e \in \text{Exp}^*$ , but  $(y, e_2) \notin \text{ExpPair}$ . However, this case can occur during the evaluation of  $(\text{letrec } H_1 \text{ in } E_1[e]) \xrightarrow{R\text{-}\{\text{garb}\}^*} (\text{letrec } H'_1 \text{ in } E_1[x])$ .

Notice that the test of the ifdead-expression failed in this case. In fact, Lemma C.6 tells us that the test of the ifdead-expression will *always* fail.

Because of the reasoning in the preceding paragraph, the unique rewrite rule that applies to  $(\text{letrec } H_2 \text{ in } E_2[E[\text{ifdead } y (e_3 e_2) e_3]])$  is

$$(\text{letrec } H_2 \text{ in } E_2[E[\text{ifdead } y (e_3 e_2) e_3]]) \xrightarrow{\text{ifdead}} (\text{letrec } H_2 \text{ in } E_2[E[e_3 w]]).$$

By the hypothesis of the Lemma we have  $H_1 \approx_{E[\text{ifdead } y (e_3 e_2) e_3]} H_2$ . Therefore we have  $H_1 \approx_{E[e_3 w]} H_2$ . Now we can apply the inductive hypothesis to get a unique reduction sequence

$$\text{letrec } H_2 \text{ in } E_2[E[e_3 w]] \xrightarrow{R\text{-}\{\text{garb}\}^*} \text{letrec } H'_2 \text{ in } E_2[x]$$

such that  $H'_1 \approx_x H'_2$ . This finishes the case.  $\square$

**Definition C.9.** Suppose  $e_1, e_2$  are expressions. The binary predicate  $\text{Eval}(e_1, e_2)$  holds if and only if for all heaps  $H_1, H_2$  such that  $H_1 \approx_{e_1} H_2$  we have:

1.  $((\text{letrec } H_1 \text{ in } E_1[e_1]) \xrightarrow{R\text{-}\{\text{garb}\}^*} (\text{letrec } H'_1 \text{ in } E_1[\text{weak } x])$  implies  $(\text{letrec } H_2 \text{ in } E_2[e_2]) \xrightarrow{R\text{-}\{\text{garb}\}^*} (\text{letrec } H'_2 \text{ in } E_2[x])$  such that  $H'_1 \approx_x H'_2$ ,

or

2.  $((\text{letrec } H_1 \text{ in } E_1[e_1]) \xrightarrow{R\text{-}\{\text{garb}\}^*} (\text{letrec } H'_1 \text{ in } E_1[\lambda x.e'_1])$  implies  $(\text{letrec } H_2 \text{ in } E_2[e_2]) \xrightarrow{R\text{-}\{\text{garb}\}^*} (\text{letrec } H'_2 \text{ in } E_2[\lambda x.e'_2])$  such that  $\text{Eval}(e'_1\{x := e\}, e'_2\{x := e\})$ .

$\square$

**Lemma C.10.** *If  $(e_1, e_2) \in \text{ExpPair}$ , then  $\text{Eval}(e_1, e_2)$ .*  $\square$

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

case: The last rule applied in the derivation is  $\frac{e \in \text{Exp}^*}{(\text{weak } e, e) \in \text{ExpPair}}$ . Then part 1. in the definition of  $\text{Eval}$  follows directly from Lemma C.8.

case: The last rule applied in the derivation is  $\frac{(e'_1, e'_2) \in \text{ExpPair}}{(\lambda x.e'_1, \lambda x.e'_2) \in \text{ExpPair}}$ .

Then the inductive hypothesis gives us  $\text{Eval}(e'_1, e'_2)$ . We can prove by induction on the definition of  $\text{Eval}(e_1, e_2)$  that if  $\text{Eval}(e_1, e_2)$  holds where  $x \in \text{FV}(e_1) \cap \text{FV}(e_2)$ , then  $\text{Eval}(e_1\{x := e\}, e_2\{x := e\})$  holds for any expression  $e$  (in this proof the base case, or part 1. in the definition of  $\text{Eval}$ , is proved by induction on the length of the reduction sequence  $(\text{letrec } H_1 \text{ in } E_1[e_1]) \xrightarrow{R\text{-}\{\text{garb}\}^*} (\text{letrec } H'_1 \text{ in } E_1[\text{weak } x])$ , using Lemma C.8). Part 2. in the definition of  $\text{Eval}$  follows from this fact.

case: The last rule applied in the derivation is  $\frac{(e'_1, e'_2) \in \text{ExpPair} \quad e_3 \in \text{Exp}^*}{(e'_1 e_3, e'_2 e_3) \in \text{ExpPair}}$ .

Then the inductive hypothesis gives us  $\text{Eval}(e'_1, e'_2)$ . If part 1. in the definition of  $\text{Eval}(e'_1, e'_2)$  holds, then the

evaluation of  $e'_1 e_3$  gets stuck, so  $\text{Eval}(e'_1 e_3, e'_2 e_3)$  holds trivially. Otherwise,  $\text{Eval}(e'_1 e_3, e'_2 e_3)$  follows easily from the conclusion of part 2. in the definition of  $\text{Eval}(e'_1, e'_2)$ .

*case:* The last rule applied in the derivation is  $\frac{(e'_1, e'_2) \in \text{ExpPair} \quad (e_3, e_4) \in \text{ExpPair}}{(\text{ifdead } e'_1 (e_3 e'_2) e_3, \text{ifdead } e'_1 (e_4 e'_2) e_4) \in \text{ExpPair}}$ . Then the inductive hypothesis gives us  $\text{Eval}(e_3, e_4)$ . By Lemma C.6, we know

$$\text{letrec } H_1 \text{ in } E_1[\text{ifdead } e'_1 (e_3 e'_2) e_3] \xrightarrow{\text{R-}\{\text{garb}\}^*} \text{letrec } H'_1 \text{ in } E_1[\text{ifdead } x (e_3 e'_2) e_3] \xrightarrow{\text{ifdead}} \text{letrec } H'_1 \text{ in } E_1[e_3 y]$$

and

$$\text{letrec } H_2 \text{ in } E_2[\text{ifdead } e'_1 (e_4 e'_2) e_4] \xrightarrow{\text{R-}\{\text{garb}\}^*} \text{letrec } H'_2 \text{ in } E_2[\text{ifdead } x (e_4 e'_2) e_4] \xrightarrow{\text{ifdead}} \text{letrec } H'_2 \text{ in } E_2[e_4 y]$$

where  $H'_1(x) = H'_2(x) = \text{weak } y$ . If part 1. in the definition of  $\text{Eval}(e_3, e_4)$  holds, then the evaluation of  $e_3 y$  gets stuck, so  $\text{Eval}(\text{ifdead } e'_1 (e_3 e'_2) e_3, \text{ifdead } e'_1 (e_4 e'_2) e_4)$  holds trivially. Otherwise,  $\text{Eval}((\text{ifdead } e'_1 (e_3 e'_2) e_3, \text{ifdead } e'_1 (e_4 e'_2) e_4))$  follows easily from the conclusion of part 2. in the definition of  $\text{Eval}(e_3, e_4)$ .  $\square$

**Lemma C.11 (Ifdead Postponement).** *Suppose  $(e_1, e_2) \in \text{ExpPair}$ . If*

$$\begin{aligned} (\text{letrec } H \text{ in } E[\text{ifdead } e_1 (e_3 e_2) e_3]) &\xrightarrow{\text{R-}\{\text{garb}\}^*} P_1 \xrightarrow{\text{garb}'} P_2 \xrightarrow{\text{add}} (\text{letrec } H' \text{ in } E[\text{ifdead } x (e_3 e_2) e_3]) \xrightarrow{\text{ifdead}} \\ &P_3 \xrightarrow{\text{R-}\{\text{garb}\}^*} (\text{letrec } H'' \text{ in } E[y z]), \end{aligned}$$

then there exists  $P'_2, P'_3, P'_4$  such that

$$P_1 \xrightarrow{\text{ifdead}} P'_2 \xrightarrow{\text{R-}\{\text{garb}\}^*} P'_3 \xrightarrow{\text{garb}'} P'_4 \xrightarrow{\text{add}} (\text{letrec } H'' \text{ in } E[y z]).$$

$\square$

*Proof.* There are a few cases to consider depending on which branch of the ifdead-expression

$$(\text{letrec } H \text{ in } E[\text{ifdead } x (e_3 e_2) e_3]) \xrightarrow{\text{ifdead}} P_3$$

chooses.

*case:* Let the following be the rewrite sequence in the hypothesis of the Lemma:

$$\begin{aligned} (\text{letrec } H \text{ in } E[\text{ifdead } e_1 (e_3 e_2) e_3]) &\xrightarrow{\text{R-}\{\text{garb}\}^*} (\text{letrec } H_1 \text{ in } E[\text{ifdead } x (e_3 e_2) e_3]) \xrightarrow{\text{garb}'} \\ (\text{letrec } H_2 \text{ in } E[\text{ifdead } x (e_3 e_2) e_3]) &\xrightarrow{\text{add}} (\text{letrec } H' \text{ in } E[\text{ifdead } x (e_3 e_2) e_3]) \xrightarrow{\text{ifdead}} \\ (\text{letrec } H' \text{ in } E[e_3 z]) &\xrightarrow{\text{R-}\{\text{garb}\}^*} (\text{letrec } H'' \text{ in } E[y z]), \end{aligned}$$

where  $H'(x) = \text{weak } z$ . Then the binding  $(x \mapsto \text{weak } y)$  could not have been inserted by (add) because  $x \in \text{FV}(\text{letrec } H_2 \text{ in } E[\text{ifdead } x (e_3 e_2) e_3])$ . So  $H_2(x) = \text{weak } y$ . Since (garb') only removes bindings from the heap  $H_1(x) = \text{weak } z$ . Therefore,

$$(\text{letrec } H_1 \text{ in } E[\text{ifdead } x (e_3 e_2) e_3]) \xrightarrow{\text{ifdead}} (\text{letrec } H_1 \text{ in } E[e_3 z]).$$

Notice  $H' \approx_{e_3} H_1$ . So, by Lemma C.8 we have

$$(\text{letrec } H_1 \text{ in } E[e_3 z]) \xrightarrow{\text{R-}\{\text{garb}\}^*} (\text{letrec } H'_1 \text{ in } E[y z]),$$

where  $H'' \approx_y H'_1$ .

Since for all  $x \in \text{FV}(E[y z])$  we have  $H'' \approx_x H'_1$ , we also have  $\text{result}(H'', E[y z]) = \text{result}(H'_1, E[y z])$ . Therefore, by Lemma C.5 we have

$$(\text{letrec } H'_1 \text{ in } E[y z]) \xrightarrow{\text{garb}'^*} (\text{letrec } H''_1 \text{ in } E[y z]) \xrightarrow{\text{add}} (\text{letrec } H'' \text{ in } E[y z]).$$

This completes the case.

*case:* If  $P_3 = (\text{letrec } H \text{ in } E[e_3 e_2])$  and  $P'_2 = (\text{letrec } H_1 \text{ in } E[e_3 e_2])$  for some heap  $H_1$ , that is both ifdead-expressions branch in the same direction (but the opposite direction from the previous case), then the proof follows almost exactly as in the previous case.

*case:* Let the following be the rewrite sequence in the hypothesis of the Lemma:

$$\begin{aligned} & (\text{letrec } H \text{ in } E[\text{ifdead } e_1 (e_3 e_2) e_3]) \xrightarrow{\text{R-}\{\text{garb}\}^*} (\text{letrec } H_1 \text{ in } E[\text{ifdead } x (e_3 e_2) e_3]) \xrightarrow{\text{garb}'^*} \\ & (\text{letrec } H_2 \text{ in } E[\text{ifdead } x (e_3 e_2) e_3]) \xrightarrow{\text{add}} (\text{letrec } H' \text{ in } E[\text{ifdead } x (e_3 e_2) e_3]) \xrightarrow{\text{ifdead}} \\ & (\text{letrec } H' \text{ in } E[e_3 e_2]) \xrightarrow{\text{R-}\{\text{garb}\}^*} (\text{letrec } H''' \text{ in } E[y e_2]) \xrightarrow{\text{R-}\{\text{garb}\}^*} (\text{letrec } H'' \text{ in } E[y z]), \end{aligned}$$

where  $H'(x) = \text{d}$ . Suppose also that

$$(\text{letrec } H_1 \text{ in } E[\text{ifdead } x (e_3 e_2) e_3]) \xrightarrow{\text{ifdead}} (\text{letrec } H_1 \text{ in } E[e_3 z]),$$

where  $H_1(x) = \text{weak } z$ . Notice  $H' \approx_{e_3} H_1$ . So, by Lemma C.8 we have

$$(\text{letrec } H_1 \text{ in } E[e_3 z]) \xrightarrow{\text{R-}\{\text{garb}\}^*} (\text{letrec } H'_1 \text{ in } E[y z]),$$

where  $H''' \approx_y H'_1$ . So we also have  $H'' \approx_y H'_1$ , since  $H''$  contains all bindings of  $H'''$ .

By Lemma C.10,  $\text{Eval}(e_1, e_2)$  holds. Notice  $H''' \approx_{e_3} H$ . When considering these two heaps, part 1. in the definition of  $\text{Eval}(e_1, e_2)$  must hold, because otherwise the reduction sequence in the hypothesis of the Lemma would become stuck. The conclusion of part 1. in the definition of  $\text{Eval}(e_1, e_2)$  tells us  $H'' \approx_z H_1$ . So we also have  $H'' \approx_z H'_1$ , since  $H'_1$  contains all bindings of  $H_1$ .

Since for all  $x \in \text{FV}(E[y z])$  we have  $H'' \approx_x H'_1$ , we also have  $\text{result}(H'', E[y z]) = \text{result}(H'_1, E[y z])$ . Therefore, by Lemma C.5 we have

$$(\text{letrec } H'_1 \text{ in } E[y z]) \xrightarrow{\text{garb}'^*} (\text{letrec } H''_1 \text{ in } E[y z]) \xrightarrow{\text{add}} (\text{letrec } H'' \text{ in } E[y z]).$$

This completes the case. □

**Lemma C.12 (Postponement).** *If*

$$P_1 \xrightarrow{\text{garb}'^*} P_2 \xrightarrow{\text{add}} P_3 \xrightarrow{\text{R-}\{\text{ifdead}\}} P_4, \quad (1)$$

*then there exists  $P'_2, P'_3$  such that*

$$P_1 \xrightarrow{\text{R-}\{\text{ifdead}\}} P'_2 \xrightarrow{\text{garb}'^*} P'_3 \xrightarrow{\text{add}} P_4. \quad (2)$$

□

*Proof.* The proof is by case analysis on elements of  $R\text{-}\{\text{garb}, \text{ifdead}\}$ . Notice that  $R\text{-}\{\text{garb}, \text{ifdead}\}$  cannot be (let-in) since let-in-expressions are not allowed in the  $\text{Exp}^*$  syntax. We show the most difficult of the remaining three cases (the cases for  $(\pi_i)$  and (app) are straightforward since neither rule alters the heap).

*case:* Reduction sequence (1) is as follows:  $(\text{letrec } H_1 \text{ in } E[hv]) \xrightarrow{\text{garb}'}$   $(\text{letrec } H_2 \text{ in } E[hv]) \xrightarrow{\text{add}}$   $(\text{letrec } H_3 \text{ in } E[hv]) \xrightarrow{\text{alloc}}$   $(\text{letrec } H_3 \uplus \{x \mapsto hv\} \text{ in } E[x])$ .

We construct reduction sequence (2).

$$(\text{letrec } H_1 \text{ in } E[hv]) \xrightarrow{\text{alloc}} (\text{letrec } H_1 \uplus \{x \mapsto hv\} \text{ in } E[x]).$$

If  $hv \neq \text{weak } y$  then

$$(\text{letrec } H_1 \uplus \{x \mapsto hv\} \text{ in } E[x]) \xrightarrow{\text{garb}'}$$

If  $hv = \text{weak } y$  then (garb') may garbage collect too much. In particular, it may bind  $x$  to  $\mathbf{d}$ . However, since (garb') can be tuned to garbage collect as much or as little as need we still have

$$(\text{letrec } H_1 \uplus \{x \mapsto hv\} \text{ in } E[x]) \xrightarrow{\text{garb}'}$$

Notice any garbage collected by (garb') in reduction sequence (1) can be collected by (garb') here because  $\text{FV}(\text{letrec } H_1 \uplus \{x \mapsto hv\} \text{ in } E[x]) \subset \text{FV}(\text{letrec } H_1 \text{ in } E[hv])$ . By the same reasoning  $\text{FV}(\text{letrec } H_2 \uplus \{x \mapsto hv\} \text{ in } E[x]) \subset \text{FV}(\text{letrec } H_2 \text{ in } E[hv])$ . So any garbage inserted by (add) in reduction sequence (1) can also be insert here. We have

$$(\text{letrec } H_2 \uplus \{x \mapsto hv\} \text{ in } E[x]) \xrightarrow{\text{add}} (\text{letrec } H_3 \uplus \{x \mapsto hv\} \text{ in } E[x]),$$

which completes the case. □

**Lemma C.13.** *If  $P$  is gc-oblivious,  $P \Downarrow_R (\text{letrec } H_1 \text{ in } x_1)$  then there exists the following reduction sequence  $P \Downarrow_{R\text{-}\{\text{garb}\}} (\text{letrec } H_2 \text{ in } x_2) \xrightarrow{\text{garb}}$   $(\text{letrec } H_3 \text{ in } x_3)$ , where  $\text{result}(H_1, x_1) = \text{result}(H_3, x_3)$ .* □

*Proof.* By induction on the number of rewrite rules in the reduction sequence  $P \Downarrow_R (\text{letrec } H_1 \text{ in } x_1)$ , using Lemma C.12, we can transform this reduction sequence into one in which garbage collection is only performed immediately before (ifdead) rewrite rules. That is, a reduction of the form,

$$P \xrightarrow{R\text{-}\{\text{garb}\}^*} P_1 \xrightarrow{\text{garb}'}$$

Consider an ifdead-expression (ifdead  $e_1$  ( $e_3$   $e_2$ )  $e_3$ ). If expression  $e_1$  contains an ifdead-expression as a subexpression, then we call this a nesting of ifdead-expressions. By induction on the nesting of ifdead-expressions, using Lemma C.11 and Lemma C.12, we can prove there exists  $P'_1$  such that

$$P \xrightarrow{R\text{-}\{\text{garb}\}^*} P'_1 \xrightarrow{\{\text{garb}', \text{add}\}^*} (\text{letrec } H_1 \text{ in } x_1). \quad (2)$$

We proceeding by induction on the nesting of ifdead-expressions because for each ifdead-expression (ifdead  $e_1$  ( $e_3$   $e_2$ )  $e_3$ ) we need:

$$(\text{letrec } H \text{ in } E[\text{ifdead } e_1 (e_3 e_2) e_3]) \xrightarrow{R\text{-}\{\text{garb}\}^*} (\text{letrec } H' \text{ in } E[\text{ifdead } (\text{weak } x) (e_3 e_2) e_3]),$$

in order to apply Lemma C.11. In other words, there can be no garbage collection while the first argument of the ifdead-expression is evaluated.

Now by Lemma C.4 applied to reduction sequence (2) there exists  $(\text{letrec } H_3 \text{ in } x_3)$  such that

$$P \xrightarrow{R\text{-}\{\text{garb}\}^*} P'_1 \xrightarrow{\text{garb}'^*} (\text{letrec } H_3 \text{ in } x_3)$$

where  $\text{result}(H_1, x_1) = \text{result}(H_3, x_3)$  and  $H_3 \subseteq H_1$ . Since  $(\text{letrec } H_1 \text{ in } x_1)$  is not reducible with respect to  $R$  and  $H_3 \subseteq H_1$ , then  $(\text{letrec } H_3 \text{ in } x_3)$  is also not reducible with respect to  $R$ . Therefore no garbage bindings exist in  $H_3$  and we have,

$$P \xrightarrow{R\text{-}\{\text{garb}\}^*} P'_1 \xrightarrow{\text{garb}} (\text{letrec } H_3 \text{ in } x_3).$$

□

**Theorem C.14 (GC-Oblivious Programs Are Well-Behaved).** *If  $P$  is gc-oblivious,  $P \Downarrow_R (\text{letrec } H_1 \text{ in } x_1)$ , and  $P \Downarrow_R (\text{letrec } H_2 \text{ in } x_2)$ , then  $\text{result}(H_1, x_1) = \text{result}(H_2, x_2)$ .* □

*Proof.* By Lemma C.13 we can transform each reduction sequence into one in which all  $(\text{garb})$  rules occur at the end. This transformation preserves the result of a program. Suppose  $P \Downarrow_R (\text{letrec } H_1 \text{ in } x_1)$  is transformed into

$$P \xrightarrow{R\text{-}\{\text{garb}\}} P_1 \xrightarrow{R\text{-}\{\text{garb}\}} P_2 \xrightarrow{R\text{-}\{\text{garb}\}} \dots \xrightarrow{R\text{-}\{\text{garb}\}} P_n \xrightarrow{\text{garb}} P_{n+1}.$$

And similarly,  $P \Downarrow_R (\text{letrec } H_2 \text{ in } x_2)$  is transformed into

$$P \xrightarrow{R\text{-}\{\text{garb}\}} P'_1 \xrightarrow{R\text{-}\{\text{garb}\}} P'_2 \xrightarrow{R\text{-}\{\text{garb}\}} \dots \xrightarrow{R\text{-}\{\text{garb}\}} P'_m \xrightarrow{\text{garb}} P'_{m+1}.$$

Then we have  $P_n = P'_m$  (in fact,  $n = m$ ) since there is a unique reduction sequences from  $P$  without garbage collection. Therefore  $P_{n+1} = P'_{m+1}$ . □

## D Proof of Conditions for Additional Garbage Collection

**Definition D.1 (Respectful Garbage Collection).** We say that a rule  $\overset{x}{\rightarrow}$  is a respectful garbage collection rule if it satisfies the following conditions.

1. For every  $\lambda_{\text{weak}}$  program  $\text{letrec } H \text{ in } e$  we have  $\text{letrec } H \text{ in } e \overset{x}{\rightarrow} \text{letrec } H' \text{ in } e$  for some heap  $H' \subseteq H$ , i.e., every  $\lambda_{\text{weak}}$  program can be reduced using  $\overset{x}{\rightarrow}$ .
2. For every  $\lambda_{\text{weak}}$  program  $P$  such that **weak** and **ifdead** do not occur in  $P$  and  $P \overset{x}{\rightarrow} P'$ , we have  $(P, R) \simeq (P', R)$ .
3. For every  $\lambda_{\text{weak}}$  program  $P$ , if  $\overset{x}{\rightarrow}$  applied to  $P$  garbage collects some binding  $x$ , then  $\overset{x}{\rightarrow}$  applied to  $\text{Trans}(P)$  also garbage collects  $x$ , i.e., the presence of weak references do not enable  $\overset{x}{\rightarrow}$  to collect extra bindings.
4. For all  $\lambda_{\text{weak}}$  programs  $P_1 = \text{letrec } H_1 \text{ in } e_1$  and  $P_2 = \text{letrec } H_2 \text{ in } e_2$  such that  $P_1 \overset{x}{\rightarrow} P_2$ , if  $\langle H_1, \text{Dom}(H_1^w), \emptyset \rangle \Rightarrow_{\text{nf}} \langle H'_1, \emptyset, H''_1 \rangle$  then  $H''_1 \subseteq H_2$ . □

**Definition D.2 (Active Free Variables).**  $x \in \text{FV}(e)$  is an active free variable of  $e$  (written  $x \in \text{AFV}(e)$ ) if and only if for all well-formed programs  $\text{letrec } H[x \mapsto hv]$  in  $e$  we have  $(\text{letrec } H[x \mapsto hv] \text{ in } e, \mathbf{R}) \not\approx (\text{letrec } H \text{ in } e, \mathbf{R})$ .

Intuitively,  $\text{AFV}(e)$  corresponds to those free variables of  $e$  that can alter the final result of the evaluation of  $e$ .  $\square$

**Definition D.3 (Active Free Variable Preservation).** We say that heaps  $H_1$  and  $H_2$  preserve the active free variables of  $e$  if for every  $x \in \text{AFV}(e)$  we have either  $H_1(x) = H_2(x)$  or  $x \notin \text{Dom}(H_1) \cup \text{Dom}(H_2)$ .  $\square$

**Lemma D.4.** If  $\overset{x}{\rightarrow}$  is a respectful garbage collection rule then for all well-formed  $\lambda_{\text{weak}}$  programs such that  $\text{letrec } H_1$  in  $e \overset{x}{\rightarrow} \text{letrec } H_2$  in  $e$ , it holds that  $H_1$  and  $H_2$  preserve the active free variables of  $e$ .  $\square$

*Proof.* Consider  $x \in \text{AFV}(e)$ . If  $x \notin \text{Dom}(H_1)$ , then  $x \notin \text{Dom}(H_2)$  by condition 1. of Definition D.1 so the lemma is satisfied. Otherwise,  $x \in \text{Dom}(H_1)$ . Now consider  $\langle H_1, \text{Dom}(H_1^w), \emptyset \rangle \Rightarrow_{\text{nf}} \langle H_1', \emptyset, H_1'' \rangle$ . If  $x \in \text{Dom}(H_1'')$  then  $H_1(x) = H_2(x)$  by condition 4. of Definition D.1. Otherwise  $x \notin \text{Dom}(H_1'')$ . Then by condition 3. of Definition D.1, if  $\overset{x}{\rightarrow}$  applied to  $\text{letrec } H_1$  in  $e$  garbage collections  $x$ , then  $\overset{x}{\rightarrow}$  applied to  $\text{Trans}(\text{letrec } H_1 \text{ in } e)$  should garbage collect  $x$ . But if  $\overset{x}{\rightarrow}$  applied to  $\text{Trans}(\text{letrec } H_1 \text{ in } e)$  garbage collects  $x$  then we have contradicted condition 2. of Definition D.1 since  $x \in \text{AFV}(e)$  and  $\text{Trans}(\text{letrec } H_1 \text{ in } e)$  is a well-formed program.  $\square$

**Lemma D.5.** Let  $\text{letrec } H_1$  in  $e$  and  $\text{letrec } H_2$  in  $e$  be  $\lambda_{\text{weak}}$  programs such that  $H_1$  and  $H_2$  preserve the active free variables of  $e$ . Also suppose  $\text{letrec } H_1$  in  $e \overset{y}{\rightarrow} \text{letrec } H_1'$  in  $e'$  where  $y \in \mathbf{R} \cup \{x\}$ . Then there exists a  $\lambda_{\text{weak}}$  program  $\text{letrec } H_2'$  in  $e'$  such that  $\text{letrec } H_2$  in  $e \overset{y}{\rightarrow} \text{letrec } H_2'$  in  $e'$ .  $\square$

*Proof.* We proceed by cases analysis on the elements of  $\mathbf{R} \cup \{x\}$ .

*case:  $y = \text{alloc}$*

Then  $\text{letrec } H_1$  in  $E[hv] \xrightarrow{\text{alloc}} \text{letrec } H_1 \uplus \{x \mapsto hv\}$  in  $E[x]$ . And  $\text{letrec } H_2$  in  $E[hv] \xrightarrow{\text{alloc}} \text{letrec } H_2 \uplus \{x \mapsto hv\}$  in  $E[x]$ .

*case:  $y = \text{proj}$*

Then  $\text{letrec } H_1$  in  $E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H_1$  in  $E[x_i]$ . Since  $H_1$  and  $H_2$  preserve the active free variables of  $E[\pi_i x]$ , we have  $H_1(x) = H_2(x)$ . So  $\text{letrec } H_2$  in  $E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H_2$  in  $E[x_i]$ .

*case:  $y = \text{app}$*

Then  $\text{letrec } H_1$  in  $E[x y] \xrightarrow{\text{app}} \text{letrec } H_1$  in  $E[e\{z := y\}]$  Since  $H_1$  and  $H_2$  preserve the active free variables of  $E[x y]$ , we have  $H_1(x) = H_2(x)$  and  $H_1(y) = H_2(y)$ . So  $\text{letrec } H_2$  in  $E[x y] \xrightarrow{\text{app}} \text{letrec } H_2$  in  $E[e\{z := y\}]$ .

*case:  $y = \text{let-in}$*

Then  $\text{letrec } H_1$  in  $E[\text{let } x = y \text{ in } e] \xrightarrow{\text{let-in}} \text{letrec } H_1$  in  $E[e\{x := y\}]$  Since  $H_1$  and  $H_2$  preserve the active free variables of  $E[\text{let } x = y \text{ in } e]$ , we have  $H_1(y) = H_2(y)$ . So  $\text{letrec } H_2$  in  $E[\text{let } x = y \text{ in } e] \xrightarrow{\text{let-in}} \text{letrec } H_2$  in  $E[e\{x := y\}]$ .

*case:  $y = \text{ifdead}$*

Then there are two subcases.

*subcase:  $\text{letrec } H_1$  in  $E[\text{ifdead } x y z] \xrightarrow{\text{ifdead}} \text{letrec } H_1$  in  $E[y]$*

Since  $H_1$  and  $H_2$  preserve the active free variables of  $E[\text{ifdead } x \ y \ z]$ , we have  $H_1(x) = H_2(x)$ . So  $\text{letrec } H_2 \text{ in } E[\text{ifdead } x \ y \ z] \xrightarrow{\text{ifdead}} \text{letrec } H_2 \text{ in } E[y]$ .

*subcase:*  $\text{letrec } H_1 \text{ in } E[\text{ifdead } x \ y \ z] \xrightarrow{\text{ifdead}} \text{letrec } H_1 \text{ in } E[z \ w]$

Since  $H_1$  and  $H_2$  preserve the active free variables of  $E[\text{ifdead } x \ y \ z]$ , we have  $H_1(x) = H_2(x)$ . So  $\text{letrec } H_2 \text{ in } E[\text{ifdead } x \ y \ z] \xrightarrow{\text{ifdead}} \text{letrec } H_2 \text{ in } E[z \ w]$ .

*case:*  $y = \text{garb}$

Then  $\text{letrec } H_1 \text{ in } E[e] \xrightarrow{\text{garb}} \text{letrec } H'_1 \text{ in } E[e]$  Since  $\xrightarrow{\text{garb}}$  always applies we have  $\text{letrec } H_2 \text{ in } E[e] \xrightarrow{\text{garb}} \text{letrec } H'_2 \text{ in } E[e]$ .

*case:*  $y = x$

Then  $\text{letrec } H_1 \text{ in } E[e] \xrightarrow{x} \text{letrec } H'_1 \text{ in } E[e]$ . Since by condition 1. of Definition D.1  $\xrightarrow{x}$  always applies we have  $\text{letrec } H_2 \text{ in } E[e] \xrightarrow{x} \text{letrec } H'_2 \text{ in } E[e]$ .  $\square$

**Lemma D.6.** *For all well-formed  $\lambda_{\text{weak}}$  programs such that  $\text{letrec } H_1 \text{ in } e \xrightarrow{y} \text{letrec } H'_1 \text{ in } e'$  and  $\text{letrec } H_2 \text{ in } e \xrightarrow{y} \text{letrec } H'_2 \text{ in } e'$  where  $y \in \mathbf{R} \cup \{x\}$ , if  $H_1$  and  $H_2$  preserve the active free variables of  $e$  then  $H'_1$  and  $H'_2$  preserve the active free variables of  $e'$ .*  $\square$

*Proof.* We proceed by cases analysis on the elements of  $\mathbf{R} \cup \{x\}$ .

*case:*  $y = \text{alloc}$

Then  $\text{letrec } H_1 \text{ in } E[hv] \xrightarrow{\text{alloc}} \text{letrec } H_1 \uplus \{x \mapsto hv\} \text{ in } E[x]$  and  $\text{letrec } H_2 \text{ in } E[hv] \xrightarrow{\text{alloc}} \text{letrec } H_2 \uplus \{x \mapsto hv\} \text{ in } E[x]$ . Since  $H_1$  and  $H_2$  preserve the active free variables of  $E[hv]$  and  $H_1(x) = H_2(x)$ , we have  $H_1 \uplus \{x \mapsto hv\}$  and  $H_2 \uplus \{x \mapsto hv\}$  preserve the active free variables of  $E[x]$ .

*case:*  $y = \text{proj}$

Then  $\text{letrec } H_1 \text{ in } E[\pi_i \ x] \xrightarrow{\pi_i} \text{letrec } H_1 \text{ in } E[x_i]$  and  $\text{letrec } H_2 \text{ in } E[\pi_i \ x] \xrightarrow{\pi_i} \text{letrec } H_2 \text{ in } E[x_i]$ . Since  $H_1$  and  $H_2$  preserve the active free variables of  $E[\pi_i \ x]$  and  $H_1(x) = H_2(x) = \langle x_1, x_2 \rangle$  implies that  $H_1(x_i) = H_2(x_i)$  we are done.

*case:*  $y = \text{app}$

Then  $\text{letrec } H_1 \text{ in } E[x \ y] \xrightarrow{\text{app}} \text{letrec } H_1 \text{ in } E[e\{z := y\}]$  and  $\text{letrec } H_2 \text{ in } E[x \ y] \xrightarrow{\text{app}} \text{letrec } H_2 \text{ in } E[e\{z := y\}]$ . Since  $H_1$  and  $H_2$  preserve the active free variables of  $E[x \ y]$  and  $H_1(x) = H_2(x) = \lambda z.e$  implies that  $H_1$  and  $H_2$  preserve the active free variables of  $e$  we are done.

*case:*  $y = \text{let-in}$

Then  $\text{letrec } H_1 \text{ in } E[\text{let } x = y \text{ in } e_2] \xrightarrow{\text{let-in}} \text{letrec } H_1 \text{ in } E[e\{x := y\}]$  and  $\text{letrec } H_2 \text{ in } E[\text{let } x = y \text{ in } e_2] \xrightarrow{\text{let-in}} \text{letrec } H_2 \text{ in } E[e\{x := y\}]$ . Since  $H_1$  and  $H_2$  preserve the active free variables of  $E[\text{let } x = y \text{ in } e]$  we are done.

*case:*  $y = \text{ifdead}$

Then there are two subcases.

*subcase:*  $\text{letrec } H_1 \text{ in } E[\text{ifdead } x \ y \ z] \xrightarrow{\text{ifdead}} \text{letrec } H_1 \text{ in } E[y]$  and  $\text{letrec } H_2 \text{ in } E[\text{ifdead } x \ y \ z] \xrightarrow{\text{ifdead}} \text{letrec } H_2 \text{ in } E[y]$

Since  $H_1$  and  $H_2$  preserve the active free variables of  $E[\text{ifdead } x \ y \ z]$  and  $y \in \text{AFV}(E[\text{ifdead } x \ y \ z])$ , we have  $H_1(y) = H_2(y)$ . So  $H_1$  and  $H_2$  preserve the active free variables of  $E[y]$ .

*subcase:*  $\text{letrec } H_1 \text{ in } E[\text{ifdead } x \ y \ z] \xrightarrow{\text{ifdead}} \text{letrec } H_1 \text{ in } E[z \ w]$  and  $\text{letrec } H_2 \text{ in } E[\text{ifdead } x \ y \ z] \xrightarrow{\text{ifdead}} \text{letrec } H_2 \text{ in } E[z \ w]$

Since  $H_1$  and  $H_2$  preserve the active free variables of  $E[\text{ifdead } x \ y \ z]$  and  $x, z \in \text{AFV}(E[\text{ifdead } x \ y \ z])$ , we have  $H_1(x) = H_2(x) = \text{weak } w$  and  $H_1(z) = H_2(z)$ . So  $H_1$  and  $H_2$  preserve the active free variables of  $E[z \ w]$ .

*case:*  $y = \text{garb}$

Then  $\text{letrec } H_1 \text{ in } e \xrightarrow{\text{garb}} \text{letrec } H'_1 \text{ in } e$  and  $\text{letrec } H_2 \text{ in } e \xrightarrow{\text{garb}} \text{letrec } H'_2 \text{ in } e$ . Since the semantics of  $\xrightarrow{\text{garb}}$  is guided by the syntax of  $e$ , if an active free variable binding is removed or altered in  $H_1$  then it will also be removed or altered in  $H_2$ . Therefore  $H'_1$  and  $H'_2$  will still preserve the active free variables of  $e$ .

*case:*  $y = x$

Then  $\text{letrec } H_1 \text{ in } e \xrightarrow{x} \text{letrec } H'_1 \text{ in } e$  and  $\text{letrec } H_2 \text{ in } e \xrightarrow{x} \text{letrec } H'_2 \text{ in } e$ . By Lemma D.4 we have  $H_1$  and  $H'_1$  preserve the active free variables of  $e$ . Also, by Lemma D.4 we have  $H_2$  and  $H'_2$  preserve the active free variables of  $e$ . Therefore we have  $H'_1$  and  $H'_2$  preserve the active free variables of  $e$ .  $\square$

**Lemma D.7.** *Let  $\xrightarrow{x}$  be a respectful garbage collection rule. Also suppose there are well-formed  $\lambda_{\text{weak}}$  programs  $\text{letrec } H \text{ in } e$  and  $\text{letrec } H' \text{ in } e$  such that  $H$  and  $H'$  preserve the active free variables of  $e$ . Then  $(\text{letrec } H \text{ in } e, R \cup \{x\}) \simeq (\text{letrec } H' \text{ in } e, R \cup \{x\})$ .*  $\square$

*Proof.* Suppose

$$\text{letrec } H \text{ in } e \xrightarrow{R \cup \{x\}} P_1 \xrightarrow{R \cup \{x\}} \dots \xrightarrow{R \cup \{x\}} P_{n-1} \xrightarrow{R \cup \{x\}} \text{letrec } H_n \text{ in } x_n.$$

Then by repeated application of Lemma D.5 and Lemma D.6 we have:

$$\text{letrec } H' \text{ in } e \xrightarrow{R \cup \{x\}} Q_1 \xrightarrow{R \cup \{x\}} \dots \xrightarrow{R \cup \{x\}} Q_{n-1} \xrightarrow{R \cup \{x\}} \text{letrec } H'_n \text{ in } x'_n.$$

Notice that as a result of Lemma D.5 and Lemma D.6 we have:

- for every  $P_i \xrightarrow{y} P_{i+1}$  we have  $Q_i \xrightarrow{y} Q_{i+1}$  where  $y \in R \cup \{x\}$  and
- for every  $P_i = \text{letrec } H_i \text{ in } e_i$  we have  $Q_i = \text{letrec } H'_i \text{ in } e_i$  it holds that  $H_i$  and  $H'_i$  preserve the active free variables of  $e_i$ .

Then  $H_n(x_n) = H'_n(x'_n)$  holds.

The opposite direction uses the same reasoning.  $\square$

**Theorem D.8 (Addition of  $\xrightarrow{x}$ ).** *If  $\xrightarrow{x}$  is a respectful garbage collection rule then for all well-formed  $\lambda_{\text{weak}}$  programs  $P$  such that  $P \xrightarrow{x} P'$  we have  $(P, R) \simeq (P', R)$ .*  $\square$

*Proof.* Let  $P = \text{letrec } H \text{ in } e$  and  $P' = \text{letrec } H' \text{ in } e$ . By Lemma D.4  $H$  and  $H'$  preserve the active free variables of  $e$ . Now by Lemma D.7 we have  $(P, R) \simeq (P', R)$ .  $\square$

## E Proof of Type Soundness

**Lemma E.1 (Inversion).**

1. If  $\Gamma \vdash x : \sigma$  then  $\Gamma(x) = \sigma$ .
2. If  $\Gamma \vdash i : \tau$  then  $\tau = \text{int}$ .
3. If  $\Gamma \vdash \langle e_1, e_2 \rangle : \tau$  then  $\tau = \tau_1 \times \tau_2$  where  $\Gamma \vdash e_1 : \tau_1$  and  $\Gamma \vdash e_2 : \tau_2$ .
4. If  $\Gamma \vdash \pi_i e : \tau$  then  $\tau = \tau_i$  where  $\Gamma \vdash e : \tau_1 \times \tau_2$  and  $i \in \{1, 2\}$ .
5. If  $\Gamma \vdash \lambda x. e : \tau$  then  $\tau = \tau_1 \rightarrow \tau_2$  for some  $\tau_1, \tau_2$  where  $\Gamma, x : \tau_1 \vdash e : \tau_2$ .
6. If  $\Gamma \vdash e_1 e_2 : \tau$  then there exists some  $\tau_1$  such that  $\Gamma \vdash e_1 : \tau_1 \rightarrow \tau$  and  $\Gamma \vdash e_2 : \tau_1$ .
7. If  $\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau$  then there exists some  $\sigma$  such that  $\Gamma \vdash e_1 : \sigma$ ,  $\Gamma \uplus \{x : \sigma\} \vdash e_2 : \tau$ .
8. If  $\Gamma \vdash \text{weak } e : \tau$  then  $\tau = \tau' \text{ weak}$  and  $\Gamma \vdash e : \tau'$ .
9. If  $\Gamma \vdash \text{d} : \tau$  then  $\tau = \tau' \text{ weak}$ .
10. If  $\Gamma \vdash \text{ifdead } e_1 e_2 e_3 : \tau$  then  $\Gamma \vdash e_1 : \tau_1 \text{ weak}$  for some  $\tau_1$  and  $\Gamma \vdash e_2 : \tau$  and  $\Gamma \vdash e_3 : \tau_1 \rightarrow \tau$ .
11. If  $\Gamma \vdash H : \Gamma'$  then for all  $x \in \text{Dom}(\Gamma')$  we have  $\Gamma \uplus \Gamma' \vdash H(x) : \Gamma'(x)$ .
12. If  $\vdash \text{letrec } H \text{ in } e : \tau$  then there exists  $\Gamma$  such that  $\emptyset \vdash H : \Gamma$  and  $\Gamma \vdash e : \tau$ . □

*Proof.* Immediate from the definition of the typing relation. □

**Lemma E.2 (Canonical Forms).**

1. If  $\emptyset \vdash H : \Gamma$  and  $\Gamma \vdash x : \text{int}$  then  $H(x) = i$ .
2. If  $\emptyset \vdash H : \Gamma$  and  $\Gamma \vdash x : \tau \text{ weak}$  then  $H(x) = \text{weak } y$  for some  $y$  or  $H(x) = \text{d}$ .
3. If  $\emptyset \vdash H : \Gamma$  and  $\Gamma \vdash x : \tau_1 \times \tau_2$  then  $H(x) = \langle x_1, x_2 \rangle$ .
4. If  $\emptyset \vdash H : \Gamma$  and  $\Gamma \vdash x : \tau_1 \rightarrow \tau_2$  then  $H(x) = \lambda x. e$ . □

*Proof.* Straightforward. By investigating the possible heap expressions and application of the Inversion Lemma. □

**Lemma E.3 (Unique Decomposition).** Any  $\lambda_{\text{weak}}$  expression  $e$  is either a variable or there exists unique  $E$  and  $I$  such that  $e = E[I]$ . □

*Proof.* By straightforward induction on  $e$ . □

**Theorem E.4 (Progress).** If  $\vdash P : \tau$  then either  $P$  is an answer or there exists  $P'$  such that  $P \xrightarrow{\text{R}} P'$ . □

*Proof.* Let  $P = \text{letrec } H \text{ in } e$ . Then by the Unique Decomposition Lemma either  $e = x$  or  $e = E[I]$ . If  $e = x$  then  $P$  is an answer. Otherwise, we proceed by case analysis on  $I$ .

*case:*  $I = hv$

Then  $\text{letrec } H \text{ in } E[hv] \xrightarrow{\text{alloc}} \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } E[x]$ .

*case:*  $I = \pi_i x$

Since  $P$  is well-typed the Inversion Lemma tells us the following must hold:  $\Gamma \vdash x : \tau_1 \times \tau_2$ . Then the Canonical Forms Lemma tells us that  $H(x) = \langle x_1, x_2 \rangle$ . So  $\text{letrec } H \text{ in } E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H \text{ in } E[x_i]$ .

*case:*  $I = x y$

Since  $P$  is well-typed the Inversion Lemma tells us the following must hold:  $\Gamma \vdash x : \tau_1 \rightarrow \tau_2$ . Then the Canonical Forms Lemma tells us that  $H(x) = \lambda z.e$ . So  $\text{letrec } H \text{ in } E[x y] \xrightarrow{\text{app}} \text{letrec } H \text{ in } E[e\{z := y\}]$ .

*case:*  $I = \text{let } x = y \text{ in } e$

Then  $\text{letrec } H \text{ in } E[\text{let } x = y \text{ in } e] \xrightarrow{\text{let-in}} \text{letrec } H \text{ in } E[e\{x := y\}]$ .

*case:*  $I = \text{ifdead } x y z$

Since  $P$  is well-typed the Inversion Lemma tells us  $\Gamma \vdash x : \tau_1$  weak. Then the Canonical Forms Lemma tells us that  $H(x) = \text{weak } w$  or  $H(x) = \mathbf{d}$ . So  $\text{letrec } H \text{ in } E[\text{ifdead } x y z] \xrightarrow{\text{ifdead}} \text{letrec } H \text{ in } E[z w]$  or  $\text{letrec } H \text{ in } E[\text{ifdead } x y z] \xrightarrow{\text{ifdead}} \text{letrec } H \text{ in } E[y]$ .  $\square$

**Lemma E.5 (Weakening).** *If  $\Gamma \vdash e : \tau$  then  $\Gamma, \Gamma' \vdash e : \tau$  provided that  $\Gamma, \Gamma'$  is a valid context.*  $\square$

*Proof.* By straightforward induction on the derivation of  $\Gamma \vdash e : \tau$ . The only case in which the context is examined is when (Var), (Fun), or (Let) is applied. It should be clear that each rule is only applicable if  $\Gamma$  contains the necessary assignment and by extending the context with additional, non-conflicting assignment we do not alter this property.  $\square$

**Lemma E.6 (Substitution).** *If  $\Gamma \vdash E[e] : \tau$  and  $\Gamma \vdash e : \tau'$  and  $\Gamma \uplus \Gamma' \vdash e' : \tau'$  then  $\Gamma \uplus \Gamma' \vdash E[e'] : \tau$ .*  $\square$

*Proof.* By induction on the typing derivation  $\Gamma \vdash E[e] : \tau$  and the Weakening Lemma.  $\square$

**Theorem E.7 (Preservation).** *If  $\vdash P : \tau$  and  $P \xrightarrow{R} P'$  then  $\vdash P' : \tau$ .*  $\square$

*Proof.* We proceed by case analysis on the rewrite rules of R.

*case:*  $\text{letrec } H \text{ in } E[hv] \xrightarrow{\text{alloc}} \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } E[x]$

Then  $\Gamma \vdash E[hv] : \tau$  and  $\Gamma \vdash hv : \tau'$  for some  $\tau, \tau', \Gamma$ . Since  $\Gamma \uplus \{x : \tau'\} \vdash x : \tau'$  by the Substitution Lemma we have  $\Gamma \uplus \{x : \tau'\} \vdash E[x] : \tau$ . Therefore  $\vdash \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } E[x] : \tau$

*case:*  $\text{letrec } H \uplus \{x \mapsto \langle x_1, x_2 \rangle\} \text{ in } E[\pi_i x] \xrightarrow{\pi_i} \text{letrec } H \uplus \{x \mapsto \langle x_1, x_2 \rangle\} \text{ in } E[x_i]$

Then  $\Gamma \vdash E[\pi_i x] : \tau$  and  $\Gamma \vdash \pi_i x : \tau_i$  for some  $\tau, \tau_i, \Gamma$ . By the Inversion Lemma  $\Gamma \vdash x : \tau_1 \times \tau_2$ . The Canonical Forms Lemma applied to the previous judgement gives us  $\Gamma \vdash H(x) : \tau_1 \times \tau_2$  where  $H(x) = \langle x_1, x_2 \rangle$ . Again by the Inversion Lemma we have  $\Gamma \vdash x_i : \tau_i$ . Finally, by the Substitution Lemma we have  $\Gamma \vdash E[x_i] : \tau$ . Then  $\vdash \text{letrec } H \uplus \{x \mapsto \langle x_1, x_2 \rangle\} \text{ in } E[x_i] : \tau$ .

*case:*  $\text{letrec } H \uplus \{x \mapsto \lambda z.e\} \text{ in } E[x y] \xrightarrow{\text{app}} \text{letrec } H \uplus \{x \mapsto \lambda z.e\} \text{ in } E[e\{z := y\}]$

Then  $\Gamma \vdash E[x y] : \tau$  and  $\Gamma \vdash x y : \tau_2$  for some  $\tau, \tau_2, \Gamma$ . By the Inversion Lemma  $\Gamma \vdash x : \tau_1 \rightarrow \tau_2$  and  $\Gamma \vdash y : \tau_1$  for some  $\tau_1$ . By the Canonical Forms Lemma we have  $\Gamma \vdash H(x) : \tau_1 \rightarrow \tau_2$  where  $H(x) = \lambda z.e$ . Again by the Inversion Lemma we have  $\Gamma \uplus \{y : \tau_1\} \vdash e\{z := y\} : \tau_2$ . Therefore by the Substitution Lemma we have  $\Gamma \uplus \{y : \tau_1\} \vdash E[e\{z := y\}] : \tau$ . Then  $\vdash \text{letrec } H \uplus \{x \mapsto \lambda z.e\} \text{ in } E[e\{z := y\}] : \tau$ , since  $\Gamma \vdash H(y) : \tau_1$ .

*case:*  $\text{letrec } H \text{ in } E[\text{let } x = y \text{ in } e] \xrightarrow{\text{let-in}} \text{letrec } H \text{ in } E[e\{x := y\}]$

Then  $\Gamma \vdash E[\text{let } x = y \text{ in } e] : \tau$  and  $\Gamma \vdash \text{let } x = y \text{ in } e : \tau'$  for some  $\tau, \tau', \Gamma$ . By the Inversion Lemma applied to the latter judgement we have  $\Gamma \vdash y : \sigma$  and  $\Gamma \uplus \{x : \sigma\} \vdash e : \tau'$  for some  $\sigma$ . Therefore by the Substitution Lemma  $\Gamma \uplus \{y : \sigma\} \vdash E[e\{x := y\}] : \tau$ . Then  $\vdash \text{letrec } H \text{ in } E[e\{x := y\}] : \tau$ , since  $\Gamma \vdash H(y) : \sigma$ .

*case:*  $\text{letrec } H \text{ in } E[\text{ifdead } x y z] \xrightarrow{\text{ifdead}} \begin{cases} \text{letrec } H \text{ in } E[z w] & \text{if } H(x) = \text{weak } w \\ \text{letrec } H \text{ in } E[y] & \text{if } H(x) = \text{d} \end{cases}$

Then  $\Gamma \vdash E[\text{ifdead } x y z] : \tau$  and  $\Gamma \vdash \text{ifdead } x y z : \tau'$  for some  $\tau, \tau', \Gamma$ . By the Inversion Lemma applied to the latter judgement we have  $\Gamma \vdash x : \tau'_1 \text{ weak}$  and  $\Gamma \vdash y : \tau'$  and  $\Gamma \vdash z : \tau'_1 \rightarrow \tau'$ . Also the Canonical Forms Lemma tells us that  $H(x) = \text{weak } w$  or  $H(x) = \text{d}$ . If  $H(x) = \text{weak } w$  then by the Inversion Lemma we have  $\Gamma \vdash w : \tau'_1$ . Therefore by the Substitution Lemma we have either  $\Gamma \vdash E[y] : \tau$  or  $\Gamma \vdash E[z w] : \tau$ . Then either  $\vdash \text{letrec } H \text{ in } E[y] : \tau$  or  $\vdash \text{letrec } H \text{ in } E[z w] : \tau$ .

*case:*  $\text{letrec } H \text{ in } e \xrightarrow{\text{garb}} \text{letrec } H' \text{ in } e$

Then  $\vdash H : \Gamma$  for some  $\Gamma$ . Notice that  $\text{Dom}(H') \subseteq \text{Dom}(H)$ , by the definition of  $\xrightarrow{\text{garb}}$ . Also, for every  $x \in \text{Dom}(H)$  either  $H'(x) = H(x)$  or  $H'(x) = \text{d}$ , by the definition of  $\xrightarrow{\text{garb}}$ . Then by typing rule (D) we have  $\Gamma \vdash H(x) : \tau'$  implies  $\Gamma \vdash H'(x) : \tau'$ . Therefore,  $\vdash \text{letrec } H' \text{ in } e : \tau$ . □

## F Proofs of Type Inference Properties

### F.1 Type Inference Soundness

**Lemma F.1.** *If  $\langle \{\tau_1 \doteq \tau_2\}, S_{Id} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S \rangle$  then  $S(\tau_1) = S(\tau_2)$ .* □

*Proof.* Immediate. By induction on the structure of  $\tau_1$ . □

**Lemma F.2.** *If  $\Gamma \vdash e : \tau$  then for any substitution  $S$  it holds that  $S(\Gamma) \vdash e : S(\tau)$ .* □

*Proof.* Immediate. By induction on the structure of the typing derivation  $\Gamma \vdash e : \tau$ . □

**Lemma F.3.** *If  $\Gamma \vdash H : \Gamma'$  then for any substitution  $S$  it holds that  $S(\Gamma) \vdash H : S(\Gamma')$ .* □

*Proof.* Immediate. By induction on the structure of the typing derivation  $\Gamma \vdash H : \Gamma'$ . □

**Lemma F.4.** *If  $\text{InferExp}(\Gamma, e) = (S, \tau)$  then  $S(\Gamma) \vdash e : \tau$ .* □

*Proof.* The proof proceeds by induction on the structure of  $e$ .

*case:*  $e = i$

Then  $\text{InferExp}(\Gamma, i) = (S_{Id}, \text{int})$  and typing rule (Int) gives us  $S_{Id}(\Gamma) \vdash i : \text{int}$ .

*case:*  $e = \text{d}$

Then  $\text{InferExp}(\Gamma, \text{d}) = (S_{Id}, \alpha \text{ weak})$  and typing rule (D) gives us  $S_{Id}(\Gamma) \vdash \text{d} : \alpha \text{ weak}$ .

*case:  $e = x$*

Then  $\text{InferExp}(\Gamma, x) = (S_{\text{Id}}, (\vec{\alpha} \mapsto \vec{\beta})(\tau))$  where  $\Gamma(x) = \forall \vec{\alpha}. \tau$  and  $\vec{\beta}$  are fresh. Typing rule (**Var**) gives us  $S_{\text{Id}}(\Gamma) \vdash x : \forall \vec{\alpha}. \tau$  and typing rule (**Inst**) applied numerous times gives  $S_{\text{Id}}(\Gamma) \vdash x : (\vec{\alpha} \mapsto \vec{\beta})(\tau)$ .

*case:  $e = \langle e_1, e_2 \rangle$*

Then  $\text{InferExp}(\Gamma, \langle e_1, e_2 \rangle) = (S_2; S_1, S_2(\tau_1) \times \tau_2)$  where  $\text{InferExp}(\Gamma, e_1) = (S_1, \tau_1)$  and  $\text{InferExp}(S_1(\Gamma), e_2) = (S_2, \tau_2)$ . By the induction hypothesis we have  $S_1(\Gamma) \vdash e_1 : \tau_1$  and  $S_2; S_1(\Gamma) \vdash e_2 : \tau_2$ . By Lemma F.2 we have  $S_2; S_1(\Gamma) \vdash e_1 : S_2(\tau_1)$ . Finally, by the application of typing rule (**Pair**) we have  $S_2; S_1(\Gamma) \vdash \langle e_1, e_2 \rangle : S_2(\tau_1) \times \tau_2$ .

*case:  $e = \pi_i e$*

Then  $\text{InferExp}(\Gamma, \pi_i e) = (S'; S, S'(\alpha_i))$  where  $\text{InferExp}(\Gamma, e) = (S, \tau)$  and  $\langle \{\tau \doteq \alpha_1 \times \alpha_2\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S' \rangle$  and  $\alpha_1, \alpha_2$  are fresh. By the induction hypothesis we have  $S(\Gamma) \vdash e : \tau$ . By Lemma F.2 we have  $S'; S(\Gamma) \vdash e : S'(\tau)$ . Then by Lemma F.1 we have  $S'(\tau) = S'(\alpha_1 \times \alpha_2)$  so  $S'; S(\Gamma) \vdash e : S'(\alpha_1 \times \alpha_2)$ . Finally, by the application of typing rule (**Proj**) we have  $S'; S(\Gamma) \vdash \pi_i e : S'(\alpha_i)$ .

*case:  $e = \lambda x. e$*

Then  $\text{InferExp}(\Gamma, \lambda x. e) = (S, S(\alpha) \rightarrow \tau)$  where  $\text{InferExp}(\Gamma \cup \{x : \alpha\}, e) = (S, \tau)$  and  $\alpha$  is fresh. By the induction hypothesis we have  $S(\Gamma \cup \{x : \alpha\}) \vdash e : \tau$ . By the application of typing rule (**Fun**) we have  $S(\Gamma) \vdash \lambda x. e : S(\alpha) \rightarrow \tau$ .

*case:  $e = e_1 e_2$*

Then  $\text{InferExp}(\Gamma, e_1 e_2) = (S'; S_2; S_1, S'(\alpha))$  where  $\text{InferExp}(\Gamma, e_1) = (S_1, \tau_1)$  and  $\text{InferExp}(S_1(\Gamma), e_2) = (S_2, \tau_2)$  and  $\langle \{S_2(\tau_1) \doteq \tau_2 \rightarrow \alpha\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S' \rangle$  and  $\alpha$  is fresh. By the induction hypothesis we have  $S_1(\Gamma) \vdash e_1 : \tau_1$  and  $S_2; S_1(\Gamma) \vdash e_2 : \tau_2$ . By Lemma F.2 we have  $S'; S_2; S_1(\Gamma) \vdash e_1 : S'; S_2(\tau_1)$  and  $S'; S_2; S_1(\Gamma) \vdash e_2 : S'(\tau_2)$ . Also by Lemma F.1 we have  $S'; S_2(\tau_1) = S'(\tau_2 \rightarrow \alpha)$ . Therefore we have  $S'; S_2; S_1(\Gamma) \vdash e_1 : S'(\tau_2 \rightarrow \alpha)$  and by application of typing rule (**App**) we have  $S'; S_2; S_1(\Gamma) \vdash e_1 e_2 : S'(\alpha)$ .

*case:  $e = \text{let } x = e_1 \text{ in } e_2$*

Then  $\text{InferExp}(\Gamma, \text{let } x = e_1 \text{ in } e_2) = (S_2; S_1, \tau_2)$  where  $\text{InferExp}(\Gamma, e_1) = (S_1, \tau_1)$  and  $\text{InferExp}(S_1(\Gamma) \cup \{x : \text{Gen}(S_1(\Gamma), \tau_1)\}, e_2) = (S_2, \tau_2)$ . By the induction hypothesis we have  $S_1(\Gamma) \vdash e_1 : \tau_1$  and  $S_2(S_1(\Gamma) \cup \{x : \text{Gen}(S_1(\Gamma), \tau_1)\}) \vdash e_2 : \tau_2$ . By Lemma F.2 we have  $S_2; S_1(\Gamma) \vdash e_1 : S_2(\tau_1)$ . Lastly, by application of the typing rule (**Gen**) we have  $S_2; S_1(\Gamma) \vdash e_1 : \text{Gen}(S_2; S_1(\Gamma), S_2(\tau_1))$  and by application of the typing rule (**Let**) we have  $S_2; S_1(\Gamma) \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2$ .

*case:  $e = \text{weak } e$*

Then  $\text{InferExp}(\Gamma, \text{weak } e) = (S, \tau \text{ weak})$  where  $\text{InferExp}(\Gamma, e) = (S, \tau)$ . By the induction hypothesis we have  $S(\Gamma) \vdash e : \tau$ . By application of the typing rule (**Weak**) we have  $S\Gamma \vdash \text{weak } e : \tau \text{ weak}$ .

*case:  $e = \text{ifdead } e_1 e_2 e_3$*

Then  $\text{InferExp}(\Gamma, \text{ifdead } e_1 e_2 e_3) = (S'_2; S_3; S_2; S'_1; S_1, S'_2; S_3(\tau_2))$  where  $\text{InferExp}(\Gamma, e_1) = (S_1, \tau_1)$  and  $\langle \{\tau_1 \doteq \alpha \text{ weak}\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S'_1 \rangle$  and  $\text{InferExp}(S'_1; S_1(\Gamma), e_2) = (S_2, \tau_2)$  and  $\text{InferExp}(S_2; S'_1; S_1(\Gamma), e_3) = (S_3, \tau_3)$  and  $\langle \{\tau_3 \doteq S_3(S_2; S'_1(\alpha) \rightarrow \tau_2)\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S'_2 \rangle$  and  $\alpha$  is fresh. By the induction hypothesis we have  $S_1(\Gamma) \vdash e_1 : \tau_1$  and  $S_2; S'_1; S_1(\Gamma) \vdash e_2 : \tau_2$  and  $S_3; S_2; S'_1; S_1(\Gamma) \vdash e_3 : \tau_3$ . By Lemma F.1 we have  $S'_1(\tau_1) = S'_1(\alpha \text{ weak})$  and  $S'_2(\tau_3) = S'_2; S_3(S_2; S'_1(\alpha) \rightarrow \tau_2)$ . By Lemma F.2

we have  $S'_2; S_3; S_2; S'_1; S_1(\Gamma) \vdash e_1 : S'_2; S_3; S_2; S'_1(\alpha \text{ weak})$  and  $S'_2; S_3; S_2; S'_1; S_1(\Gamma) \vdash e_2 : S'_2; S_3(\tau_2)$  and  $S'_2; S_3; S_2; S'_1; S_1(\Gamma) \vdash e_3 : S'_2; S_3(S_2; S'_1(\alpha) \rightarrow \tau_2)$ . Due to the  $\alpha$  is fresh condition we have  $S'_2; S_3; S_2; S'_1; S_1(\Gamma) \vdash e_1 : S'_2; S_3(\alpha \text{ weak})$ . Finally, by application of the typing rule (Ifdead) we have  $S'_2; S_3; S_2; S'_1; S_1(\Gamma) \vdash \text{ifdead } e_1 e_2 e_3 : S'_2; S_3(\tau_2)$ .  $\square$

**Lemma F.5.** *If  $\text{InferHeap}(\Gamma, H) = \Gamma'$  then  $\vdash H : \Gamma'$ .*  $\square$

*Proof.* We proceed by induction on the size of the heap.

*Base case:*

Suppose the size of  $H$  is one. Then  $\text{InferHeap}(\Gamma, \{x \mapsto hv\}) = S'; S(\Gamma)$  where  $\text{InferExp}(\Gamma, hv) = (S, \tau)$  and  $\langle \{S\Gamma(x) \doteq \tau\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S' \rangle$ . By Lemma F.4 we have  $S(\Gamma) \vdash hv : \tau$ . Notice that Lemma F.1 gives us  $S'; S(\Gamma(x)) = S'(\tau)$ . Therefore Lemma F.2 gives  $S'; S(\Gamma) \vdash hv : S'; S(\Gamma(x))$  and typing rule (Heap) yeilds  $\vdash H : S'; S(\Gamma)$ .

*Inductive case:*

Suppose the size of  $H = H' \uplus \{x \mapsto hv\}$  is  $n$ . Then  $\text{InferHeap}(\Gamma, H' \uplus \{x \mapsto hv\}) = \text{InferHeap}(S'; S(\Gamma), H')$  where  $\text{InferExp}(\Gamma, hv) = (S, \tau)$  and  $\langle \{S(\Gamma(x)) \doteq \tau\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S' \rangle$ . By the induction hypothesis we have  $\vdash H' : \Gamma'$  where  $\text{InferHeap}(S'; S(\Gamma), H') = \Gamma'$ . By Lemma F.4 we have  $S(\Gamma) \vdash hv : \tau$ . Notice that Lemma F.1 gives us  $S'; S(\Gamma(x)) = S'(\tau)$ . Therefore Lemma F.2 gives  $S'; S(\Gamma) \vdash hv : S'; S(\Gamma(x))$ . Lastly, by the definition of the algorithm **InferHeap** there exists some substitution  $S''$  such that  $S''; S'; S(\Gamma) = \Gamma'$ , so Lemma F.2 gives us  $\Gamma' \vdash hv : \Gamma'(x)$  and typing rule (Heap) yeilds  $\vdash H : \Gamma'$ .  $\square$

**Theorem F.6 (Soundness of Infer).** *For any  $\lambda_{\text{weak}}$  program  $P$ , if  $\text{Infer}(P) = \tau$  then  $\vdash P : \tau$ .*  $\square$

*Proof.* Let  $P = \text{letrec } H \text{ in } e$ . Assume  $\text{InferHeap}(\Gamma(H), H) = \Gamma$ . Then Lemma F.5 gives us  $\vdash H : \Gamma$ . Assume  $\text{InferExp}(\Gamma, e) = (S, \tau)$ . Then Lemma F.4 gives us  $S(\Gamma) \vdash e : \tau$ . Lemma F.3 gives  $\vdash H : S(\Gamma)$ . Finally, by application of typing rule (Prog) we get  $\vdash P : \tau$ .  $\square$

## F.2 Type Inference Completeness

**Definition F.7.** We say that  $\Gamma$  is stronger than  $\Gamma'$ , written  $\Gamma \succeq \Gamma'$ , if  $\Gamma$  as a function extends  $\Gamma'$ , i.e, for all variables  $x$  if  $\Gamma'(x)$  is defined then  $\Gamma(x)$  is also defined and  $\Gamma'(x) = \Gamma(x)$ .  $\square$

**Lemma F.8.** *If substitution  $S$  unifies  $\tau_1 \doteq \tau_2$  and  $\langle \{\tau_1 \doteq \tau_2\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, S' \rangle$  then there exists a substitution  $T$  such that  $T; S' = S$ .*  $\square$

*Proof.* Immediate from the definition of  $\xrightarrow{\text{unify}^*}$ .  $\square$

**Lemma F.9.** *Suppose  $\Gamma' \vdash e : \tau$  and  $S(\Gamma) \succeq \Gamma'$  for some substitution  $S$ . Then  $\text{InferExp}(\Gamma, e) = (S', \tau')$  and there exists some substitution  $T$  such that:*

1.  $S = T; S'$  except on the new type variables introduced by **InferExp** and
2.  $T(\tau') = \tau$ .  $\square$

*Proof.* We proceed by induction on the structure of  $e$ .

*case:  $e = i$*

Then  $\tau = \text{int}$  and  $\text{InferExp}(\Gamma, i) = (S_{\text{Id}}, \text{int})$ . We have  $S(\text{int}) = \text{int}$  and  $S = S_{\text{Id}}; S$ .

*case:  $e = d$*

Then  $\tau = \tau'$  weak and  $\text{InferExp}(\Gamma, d) = (S_{\text{Id}}, \alpha \text{ weak})$ , where  $\alpha$  is fresh. Consider the substitution  $S[\alpha \mapsto \tau']$ . Then we have  $S[\alpha \mapsto \tau'](\alpha \text{ weak}) = \tau' \text{ weak}$  and  $S = S_{\text{Id}}; S[\alpha \mapsto \tau']$  except on the new type variables introduced by  $\text{InferExp}$ .

*case:  $e = x$*

Then  $\text{InferExp}(\Gamma, x) = (S_{\text{Id}}, (\vec{\alpha} \mapsto \vec{\beta})(\tau'))$  where  $\Gamma(x) = \forall \vec{\alpha}. \tau'$  and  $\vec{\beta}$  are fresh. By  $S(\Gamma) \succeq \Gamma'$  we have  $S(\Gamma) \vdash x : \tau$ . Without a loss of generality, we may assume this derivation has the following form:

$$\frac{\frac{S(\Gamma(x)) = \forall \vec{\alpha}. \tau' \quad (\text{Var})}{S(\Gamma) \vdash x : \forall \vec{\alpha}. \tau'} \quad (\text{Inst})}{S(\Gamma) \vdash x : (\vec{\alpha} \mapsto \vec{\gamma})(\tau')}$$

where  $\tau = (\vec{\alpha} \mapsto \vec{\gamma})(\tau')$ . Consider the substitution  $S[\vec{\beta} \mapsto \vec{\gamma}]$ . Then we have  $S[\vec{\beta} \mapsto \vec{\gamma}](\vec{\alpha} \mapsto \vec{\beta})(\tau') = S((\vec{\alpha} \mapsto \vec{\gamma})(\tau')) = \tau$ . Also  $S = S_{\text{Id}}; S[\vec{\beta} \mapsto \vec{\gamma}]$  except on the new type variables introduced by  $\text{InferExp}$ .

*case:  $e = \langle e_1, e_2 \rangle$*

Then  $\tau = \tau_1 \times \tau_2$ . Also,  $\text{InferExp}(\Gamma, \langle e_1, e_2 \rangle) = (T_2; T_1, T_2(\bar{\tau}_1) \times \bar{\tau}_2)$  where  $\text{InferExp}(\Gamma, e_1) = (T_1, \bar{\tau}_1)$  and  $\text{InferExp}(T_1(\Gamma), e_2) = (T_2, \bar{\tau}_2)$ . Without a loss of generality we can assume that the derivation of  $\Gamma' \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2$  ends with:

$$\frac{\Gamma' \vdash e_1 : \tau_1 \quad \Gamma' \vdash e_2 : \tau_2 \quad (\text{Pair})}{\Gamma' \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

By the induction hypothesis there exists a substitution  $R$  such that  $S = R; T_1$  and  $R(\bar{\tau}_1) = \tau_1$ . Again by the induction hypothesis we know there exists a substitution  $R'$  such that  $R = R'; T_2$  and  $R'(\bar{\tau}_2) = \tau_2$ . Therefore,  $S = R'; T_2; T_1$  and  $R'(T_2(\bar{\tau}_1) \times \bar{\tau}_2) = \tau_1 \times \tau_2$ .

*case:  $e = \pi_i e$*

Then  $\text{InferExp}(\Gamma, \pi_i e) = (T'; T, T'(\alpha_i))$  where  $\text{InferExp}(\Gamma, e) = (T, \bar{\tau})$  and  $\langle \{\bar{\tau} \doteq \alpha_1 \times \alpha_2\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, T' \rangle$  and  $\alpha_1, \alpha_2$  are fresh. Without a loss of generality we can assume that the derivation of  $\Gamma' \vdash \pi_i e : \tau$  ends with:

$$\frac{\Gamma' \vdash e : \tau_1 \times \tau_2 \quad (\text{Proj})}{\Gamma' \vdash \pi_i e : \tau_i}$$

By the induction hypothesis there exists a substitution  $R$  such that  $S = R; T$  and  $R(\bar{\tau}) = \tau_1 \times \tau_2$ . Consider substitution  $R[\alpha_i \mapsto \bar{\tau}'_i]$ . Since this substitution unifies  $\bar{\tau} \doteq \alpha_1 \times \alpha_2$ , by Lemma F.8 there exists a substitution  $R'$  such that  $R[\alpha_i \mapsto \bar{\tau}'_i] = R'; T'$ . Therefore  $S = R'; T'; T$  except on the new type variables introduced by  $\text{InferExp}$  and  $R'; T'(\alpha_i) = \tau_i$ .

*case:  $e = \lambda x. e$*

Then  $\tau = \tau_1 \rightarrow \tau_2$ . Also,  $\text{InferExp}(\Gamma, \lambda x. e) = (T, T(\alpha) \rightarrow \bar{\tau})$  where  $\text{InferExp}(\Gamma \cup \{x : \alpha\}, e) = (T, \bar{\tau})$  and  $\alpha$  is fresh. Without a loss of generality we can assume that the derivation of  $\Gamma' \vdash \lambda x. e : \tau_1 \rightarrow \tau_2$  ends with:

$$\frac{\Gamma' \uplus \{x : \tau_1\} \vdash e : \tau_2}{\Gamma' \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \text{ (Fun)}$$

Notice that  $S(\Gamma) \succeq \Gamma'$  implies  $S[\alpha \mapsto \tau_1](\Gamma \cup \{x : \alpha\}) \succeq \Gamma' \cup \{x : \tau_1\}$ . By the induction hypothesis there exists a substitution  $R$  such that  $S[\alpha \mapsto \tau_1] = R; T$  and  $R(\bar{\tau}) = \tau_2$ . Notice also that  $R; T(\alpha) = \tau_1$ . So  $R(T(\alpha) \rightarrow \bar{\tau}) = \tau_1 \rightarrow \tau_2$ .

*case:  $e = e_1 e_2$*

Then  $\text{InferExp}(\Gamma, e_1 e_2) = (T'; T_2; T_1, T'(\alpha))$  where  $\text{InferExp}(\Gamma, e_1) = (T_1, \bar{\tau}_1)$  and  $\text{InferExp}(T_1(\Gamma), e_2) = (T_2, \bar{\tau}_2)$  and  $\langle \{T_2(\bar{\tau}_1) \doteq \bar{\tau}_2 \rightarrow \alpha\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, T' \rangle$  and  $\alpha$  is fresh. Without a loss of generality we can assume that the derivation of  $\Gamma' \vdash e_1 e_2 : \tau$  ends with:

$$\frac{\Gamma' \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma' \vdash e_2 : \tau_1}{\Gamma' \vdash e_1 e_2 : \tau_2} \text{ (App)}$$

By the induction hypothesis there exists a substitution  $R$  such that  $S = R; T_1$  and  $R(\bar{\tau}_1) = \tau_1 \rightarrow \tau_2$ . Also by the induction hypothesis there exists a substitution  $R'$  such that  $R = R'; T_2$  and  $R'(\bar{\tau}_2) = \tau_1$ . Consider substitution  $R'[\alpha \mapsto \tau_2]$ . Since this substitution unifies  $T_2(\bar{\tau}_1) \doteq \bar{\tau}_2 \rightarrow \alpha$ , by Lemma F.8 there exists a substitution  $R''$  such that  $R'[\alpha \mapsto \tau_2] = R''; T'$ . Therefore  $S = R''; T'; T_2; T_1$  except on the new type variables introduced by  $\text{InferExp}$  and  $R''; T'(\alpha) = \tau_2$ .

*case:  $e = \text{let } x = e_1 \text{ in } e_2$*

Then  $\text{InferExp}(\Gamma, \text{let } x = e_1 \text{ in } e_2) = (T_2; T_1, \bar{\tau}_2)$  where  $\text{InferExp}(\Gamma, e_1) = (T_1, \bar{\tau}_1)$  and  $\text{InferExp}(T_1(\Gamma) \cup \{x : \text{Gen}(T_1(\Gamma), \bar{\tau}_1)\}, e_2) = (T_2, \bar{\tau}_2)$ . Without a loss of generality we can assume that the derivation of  $\Gamma' \vdash \text{let } x = e_1 \text{ in } e_2 : \tau$  ends with:

$$\frac{\Gamma' \vdash e_1 : \sigma \quad \Gamma' \cup \{x : \sigma\} \vdash e_2 : \tau}{\Gamma' \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \text{ (Let)}$$

Since  $\Gamma' \vdash e_1 : \sigma$  there exists a  $\tau''$  such that  $\Gamma' \vdash e_1 : \tau''$ . Then by the induction hypothesis there exists a substitution  $R$  such that  $S = R; T_1$  and  $R(\bar{\tau}_1) = \tau''$ .

In order to use the induction hypothesis again we must show  $R(T_1(\Gamma) \cup \{x : \text{Gen}(T_1(\Gamma), \bar{\tau}_1)\}) \succeq \Gamma' \cup \{x : \sigma\}$ . To see this notice that for any  $\rho$  such that  $\Gamma' \cup \{x : \sigma\} \vdash x : \rho$  we have  $\Gamma' \vdash e_1 : \rho$ . Then by the induction hypothesis there exists a substitution  $V$  such that  $S = V; T_1$  and  $V(\bar{\tau}_1) = \rho$ . Notice that  $V = R$ . Then we can apply typing rule (Inst) enough times so that  $R(T_1(\Gamma) \cup \{x : \text{Gen}(T_1(\Gamma), \bar{\tau}_1)\}) \vdash x : R(\bar{\tau}_1)$ .

Now by the induction hypothesis there exists a substitution  $R'$  such that  $R = R'; T_2$  and  $R'(\bar{\tau}_2) = \tau$ .

*case:  $e = \text{weak } e$*

Then  $\tau = \tau_1 \text{ weak}$ . Also,  $\text{InferExp}(\Gamma, \text{weak } e) = (T, \bar{\tau} \text{ weak})$  where  $\text{InferExp}(\Gamma, e) = (T, \bar{\tau})$ . Without a loss of generality we can assume that the derivation of  $\Gamma' \vdash \text{weak } e : \tau_1 \text{ weak}$  ends with:

$$\frac{\Gamma' \vdash e : \tau}{\Gamma' \vdash \text{weak } e : \tau \text{ weak}} \text{ (Weak)}$$

Then by the induction hypothesis there exists a substitution  $R$  such that  $S = R; T$  and  $R(\bar{\tau}) = \tau$ . Finally,  $R(\bar{\tau} \text{ weak}) = \tau \text{ weak}$ .

*case:  $e = \text{ifdead } e_1 e_2 e_3$*

Then  $\text{InferExp}(\Gamma, \text{ifdead } e_1 e_2 e_3) = (T'_2; T_3; T_2; T'_1; T_1, T'_2; T_3(\bar{\tau}_2))$  where  $\text{InferExp}(\Gamma, e_1) = (T_1, \bar{\tau}_1)$  and  $\langle \{\bar{\tau}_1 \doteq \alpha \text{ weak}\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, T'_1 \rangle$  and  $\text{InferExp}(T'_1; T_1(\Gamma), e_2) = (T_2, \bar{\tau}_2)$  and  $\text{InferExp}(T_2; T'_1; T_1(\Gamma), e_3) = (T_3, \bar{\tau}_3)$

and  $\langle \{\bar{\tau}_3 \doteq T_3(T_2; T'_1(\alpha) \rightarrow \bar{\tau}_2)\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, T'_2 \rangle$  and  $\alpha$  is fresh. Without a loss of generality we can assume that the derivation of  $\Gamma' \vdash \text{ifdead } e_1 e_2 e_3 : \tau$  ends with:

$$\frac{\Gamma \vdash e_1 : \tau_1 \text{ weak} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau_1 \rightarrow \tau}{\Gamma \vdash \text{ifdead } e_1 e_2 e_3 : \tau} \text{ (Ifdead)}$$

By the induction hypothesis there exists a substitution  $R_1$  such that  $S = R_1; T_1$  and  $R_1(\bar{\tau}_1) = \tau_1 \text{ weak}$ . Consider substitution  $R_1[\alpha \mapsto \tau_1]$ . Since this substitution unifies  $\bar{\tau}_1 \doteq \alpha \text{ weak}$  by Lemma F.8 there exists a substitution  $R'_1$  such that  $R_1[\alpha \mapsto \tau_1] = R'_1; T'_1$ . By the induction hypothesis there exists a substitution  $R_2$  such that  $R'_1 = R_2; T_2$  and  $R_2(\bar{\tau}_2) = \tau$ . Again by the induction hypothesis there exists a substitution  $R_3$  such that  $R_2 = R_3; T_3$  and  $R_3(\bar{\tau}_3) = \tau_1 \rightarrow \tau$ . Since  $R_3$  unifies  $\bar{\tau}_3 \doteq T_3(T_2; T'_1(\text{alpha}) \rightarrow \bar{\tau}_2)$ , by Lemma F.8 there exists a substitution  $R'_2$  such that  $R_3 = R'_2; T'_2$ . Therefore  $S = R'_2; T'_2; T_3; T_2; T'_1; T_1$  except on the new type variables introduced by **InferExp** and  $R'_2; T'_2; T_3(\bar{\tau}_2) = \tau$ .  $\square$

**Lemma F.10.** *If  $\emptyset \vdash H : \Gamma'$  and  $S(\Gamma) \succeq \Gamma'$  then  $\text{InferHeap}(\Gamma, H) = \Delta$  and there exists some substitution  $S'$  such that  $S'(\Delta) \succeq \Gamma'$ .*  $\square$

*Proof.* We proceed by induction on the size of  $H$ .

*Base case:*

Suppose the size of  $H$  is one. Then  $\text{InferHeap}(\Gamma, \{x \mapsto hv\}) = T'; T(\Gamma)$  where  $\text{InferExp}(\Gamma, hv) = (T, \tau)$  and  $\langle \{T(\Gamma(x)) \doteq \tau\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, T' \rangle$ . By inversion of the typing rule (**Heap**) we have  $\Gamma' \vdash hv : \Gamma'(x)$ . By Lemma F.9 there exists a substitution  $R$  such that  $R(\tau) = \Gamma'(x)$  and  $S = R; T$ . Notice that  $S(\Gamma(x)) = \Gamma'(x)$  by the definition of  $S(\Gamma) \succeq \Gamma'$ . Now notice that  $R$  unifies  $T(\Gamma(x)) \doteq \tau$  because  $R; T(\Gamma(x)) = S(\Gamma(x)) = \Gamma'(x) = R(\tau)$ . Therefore, by Lemma F.8 we have  $R = R'; T'$  for some substitution  $R'$ . Finally we have  $S(\Gamma) = R'; T'; T(\Gamma)$  and  $R'; T'; T(\Gamma) \succeq \Gamma'$ .

*Inductive case:*

Suppose the size of  $H = H' \uplus \{x \mapsto hv\}$  is  $n$ . Then  $\text{InferHeap}(\Gamma, H' \uplus \{x \mapsto hv\}) = \text{InferHeap}(T'; T(\Gamma), H')$  where  $\text{InferExp}(\Gamma, hv) = (T, \tau)$  and  $\langle \{T(\Gamma(x)) \doteq \tau\}, S_{\text{Id}} \rangle \xrightarrow{\text{unify}^*} \langle \emptyset, T' \rangle$ . By the same reasoning as in the base case we have  $T'; T(\Gamma) \succeq \Gamma'$ . And the induction hypothesis gives us our result.  $\square$

**Theorem F.11 (Completeness and Principality of Infer).** *For any  $\lambda_{\text{weak}}$  program  $P$ , if  $\vdash P : \tau$  then  $\text{Infer}(P) = \tau'$  and there exists a substitution  $S$  such that  $S(\tau') = \tau$ .*  $\square$

*Proof.* Let  $P = \text{letrec } H \text{ in } e$ . Then by inversion of the typing rule (**Prog**) we have  $\emptyset \vdash H : \Gamma$  and  $\Gamma \vdash e : \tau$ . Then  $\text{Infer}(\text{letrec } H \text{ in } e) = \tau'$  where  $\text{InferHeap}(\Gamma(H), H) = \Delta$  and  $\text{InferExp}(\Delta, e) = \tau'$ . Since there exists substitution  $S$  such that  $S(\Gamma(H)) \succeq \Gamma$  Lemma F.10 tells us that  $\text{InferHeap}(\Gamma(H), H)$  is defined and there exists some substitution  $S'$  such that  $S'(\Delta) \succeq \Gamma$ . Lastly, Lemma F.9 tells us that  $\text{InferExp}(\Delta, e)$  is defined and there exists some substitution  $T$  such that  $T(\tau') = \tau$ .  $\square$