Closure Conversion Is Safe for Space

ZOE PARASKEVOPOULOU, Princeton University, USA
ANDREW W. APPEL, Princeton University, USA

We formally prove that closure conversion with flat environments for CPS lambda calculus is correct (preserves semantics) and safe for time and space, meaning that produced code preserves the time and space required for the execution of the source program.

We give a cost model to pre- and post-closure-conversion code by formalizing profiling semantics that keep track of the time and space resources needed for the execution of a program, taking garbage collection into account. To show preservation of time and space we set up a general, "garbage-collection compatible", binary logical relation that establishes invariants on resource consumption of the related programs, along with functional correctness. Using this framework, we show semantics preservation and space and time safety for terminating source programs, and divergence preservation and space safety for diverging source programs.

This is the first formal proof of space-safety of a closure-conversion transformation. The transformation and the proof are parts of the CertiCoq compiler pipeline from Coq (Gallina) through CompCert Clight to assembly language. Our results are mechanized in the Coq proof assistant.

CCS Concepts: • **Software and its engineering** → **General programming languages**.

Additional Key Words and Phrases: compiler correctness, closure conversion, continuation-passing style, garbage collection, logical relations, cost models

ACM Reference Format:
Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure Conversion Is Safe for Space. *Proc. ACM Program. Lang.* 3, ICFP, Article 83 (August 2019), 29 pages. https://doi.org/10.1145/3341687

1 INTRODUCTION

Formally verified compilers [Kumar et al. 2014; Leroy 2009; Neis et al. 2015] guarantee that the compiled executable behaves according to the specification of the source language. Most of the times, however, this specification is limited to the result of the computation, *i.e.* the extensional behavior of the program. But programmers also expect compilers to preserve programs' *intensional* properties, such as resource consumption, and failure to do so may result in performance and security leaks. At the same time, static cost analysis frameworks enable programmers to formally reason about the running time [Hoffmann et al. 2017; Jost et al. 2010; Wang et al. 2017] and memory usage [Albert et al. 2010; Hoffmann et al. 2017; Unnikrishnan and Stoller 2009] of programs. But a compiler that fails to preserve resource consumption renders source-level cost analysis useless. There are few examples in the literature of program transformations certified with respect to resource consumption [Crary and Weirich 2000; Minamide 1999], and most are limited to running time. We develop a general proof framework, based on logical relations, that supports reasoning about preservation of resource consumption. Inspired by a well-known example of space-safety failure—the (still-employed by implementations of functional languages) linked closure conversion

Authors' addresses: Zoe Paraskevopoulou, Princeton University, USA, zoe.paraskevopoulou@princeton.edu; Andrew W. Appel, Princeton University, USA, appel@princeton.edu.

2475-1421/2019/8-ART83
https://doi.org/10.1145/3341687

algorithm—we apply our framework to show that flat closure conversion is safe with respect to both time and space.

Closure conversion is used to implement static scoping in languages with nested functions: a program with nested lambdas that may reference variables nonlocal to their definition is transformed to a flat-scope program in which lambdas do not have free variables, and are packaged together with their environment, that contains the values of their free variables, to form a *closure*. Designers of optimizing compilers try to optimize the closure data structures for creation time, access depth of variables, and space usage, and a standard optimization technique is to share parts of the closure environment across multiple closures. If a closure, however, contains variables of different future lifetimes (some of which may be pointers to large data structures) then the garbage collector cannot reclaim the data until the entire closure-pair is no longer accessible; this can increase the program's memory use by an amount *not bounded by any constant factor* [Appel 1992]. Closure conversion that does not increase the space (respectively, time) usage of a garbage-collected program (by more than a constant factor per program) is called *safe for space* (respectively, safe for time). In fact, shared environment representations that may leak space are still employed: the JavaScript V8 engine's environment sharing strategy, based on linked environment representations, is not safe for space [Egorov 2012]. Standard flat closures are safe for space, but not necessarily optimal for creation time. More efficient safe-for-space closure conversion algorithms exist [Shao and Appel 1994, 2000], but no one has attempted to formally reason about space safety of closure conversion.

The difficulty is, to reason formally about space consumption in garbage-collected programs, it is not enough to account for the allocated heap cells during execution. One must explicitly reason about the number of cells simultaneously live in the heap at any point during the execution of the source program, and show that this is preserved by the transformed program. Minamide [1999] employs this technique to prove space preservation of the CPS transformation. Using a simulation argument, he shows that the maximum size of the reachable heap, *i.e.* the ideal amount of space required for a program's execution, is preserved by the transformation. In closure conversion this is more challenging as it changes the shape of a program's heap data structures and lifetime much more than CPS conversion. Furthermore, we are aiming at not just showing that the ideal space usage is preserved, but also connecting the idealized space usage (reflected in our source cost model) with a target cost model that is closer to the actual implementation. We achieve that that by accounting for the size of the whole heap (not just its reachable part) and explicitly modeling calls to the garbage collector.

Our proof uses a standard technique in proving semantic preservation: logical relations. The main technical novelty is that in our logical relation we impose pre- and postconditions on the related programs. We use them to establish the (time and space) resource bounds simultaneously with functional correctness, by showing that the input and output programs of the transformation inhabit the logical relation. We overcome several technical difficulties associated with logical relations. The presence of garbage collection complicates Kripke monotonicity, which states that whenever two values are related, they remain related for future states of a program's execution. Invoking a garbage collector causes heaps to not only grow during execution, but also shrink and become renamed (in the case of copying garbage collectors). We overcome this by explicitly quantifying over all future heaps in our logical relation definitions (as is common in Kripke logical relations) for the right notion of "future". We also explain why pre- and postcondition monotonicity (which in Hoare logic enables compositional reasoning with the use of weakening, strengthening, and frame rules) does not hold directly for our logical relations, and how it is possible to restore it. Finally, we show how our logical relation can be used to prove that divergence and space consumption of diverging programs is preserved (a program can run indefinitely in a bounded memory).

Contributions.

- Our general framework, based on step-indexed logical relations, establishes intensional relational properties of programs simultaneously with semantic preservation. This technique is compatible with both garbage collection and divergent source programs, two aspects that cannot be handled by traditional logical relations and require complex machinery.
- We define time and space profiling semantics for CPS λ-calculus, for code before and after closure conversion. Our "idealized" source cost model is appropriate for programmers to reason about the consumption of their programs, whereas the target cost model is closer to the implementation of the language.
- With these tools, we formally prove that flat closure conversion is correct and safe for time and space. To the best of our knowledge, this is the first formal proof of space-safety of a closure conversion transformation. Our results are all mechanized in the Coq proof assistant. The CPS intermediate representation and the closure conversion transformation are parts of the CertiCoq compiler [Anand et al. 2017], an extraction pipeline from Coq to C. [1]

All the phases of CertiCoq—from Coq abstract-syntax trees to assembly language—are verified for functional correctness, or in the process of being verified. Our closure-conversion correctness proof composes with correctness proofs of the other phases.[2]

The rest of the paper is structured as follows. In section 2 we give an overview of closure conversion and different closure environment representations, explaining why linked environments fail to preserve asymptotic complexity. In sections 5 and 6 we give the formal definition of the language, its semantics, and the closure-conversion transformation. In section 7, we describe our logical relation framework and in section 8 we apply it to prove correctness of the closure conversion transformation. We conclude with related and future work (sections 9 and 10).

2 CLOSURE REPRESENTATION

In functional languages like ML, nested functions can access variables that are nonlocal to their definition but are formal parameters or local definitions of an enclosing function. To implement accesses to nonlocal variables, compilers employ a closure-conversion transformation [Appel and Jim 1989; Kelsey and Hudak 1989], in which the environment of each function, represented as a record, is passed as an extra parameter and free-variable accesses are compiled to accesses to the environment parameter. Function values are represented as *closures*, i.e. pairs of a code pointer and the environment. At application time the code and the environment components are projected out of the pair and the latter is passed as an argument to the former.

The representation of closure environments is crucial to the design of a closure conversion algorithm. Several closure representations have been proposed, each of them trying to optimize metrics such as space consumption, number of accesses to the environment, and closure creation time [Keep et al. 2012; Shao and Appel 1994, 2000]. However, the choice of representation may affect the space-safety of a program.

2.1 Flat Closure Representation

In the *flat* representation of closures (Figure 1b), the environment of each function is a record that contains exactly the values of the function's free variables.

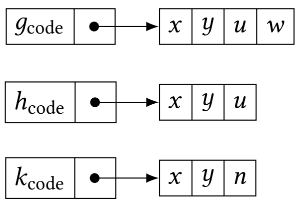
Execution time. The time overhead of flat closure conversion consists of the time needed to allocate the closure environment and pair upon each function definition, which is proportional

[1]Our development is also available as a standalone artifact (https://github.com/zoep/safe-for-space).

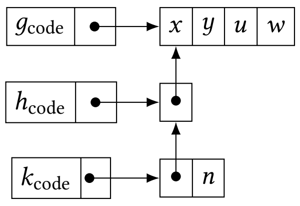
[2]The other phases are not (yet) proved safe-for-space; the framework outlined in this paper shows one method by which that could be done.

```
let f x y u w =
  let g () =
    let h n =
      let k m  =
        x + y + n
      in k u  +  n
    in h w
  in g
```

(a) Program with nested scopes



(b) Flat-closure representation



(c) Linked-closure representation

Fig. 1. Flat and linked closure representations. Linked closures *appear* to save space; for example, the flat closure environment for h is three words (x, y, u) but the linked representation is just one word. But linked closures are not safe for space; suppose k is live but g is not, then with flat closures w is garbage-collectible but with linked closures w is still reachable. If w is the root of a large data structure, this is significant.

to some constant amount plus the size of the function's environment, and the time needed for fetching free-variable values from the environment, which in flat environments is always constant. Since the size of the environment is the number of free variables in the function's body, the total overhead cannot exceed the consumption of the source times a factor proportional to the size of the source program.

In the above analysis, we implicitly assume that, in the source cost model, function definitions incur constant cost. As we will describe in section 5, in our source cost model, function definitions incur time proportional to the number of their free variables, that allows us to establish a more precise bound on the running time of the target.

Execution space. To reason about the space overhead, we must think about the amount of live space in the heap of the closure converted program. For each function definition, the code explicitly constructs an environment and stores it in the heap, which takes up space proportional to the number of free variables in the function. But also, storing the values of free variables in the environment record keeps them alive during the execution of the function for at least as long as the environment pointer is live, which is at most when the function exits. This might be past the variable's original lifetime, i.e. its last use within its scope, and therefore it prevents the garbage collector from reclaiming the space. Notice, however, that the lifetime of any variable cannot become extended beyond the lifetime of the function in which it was originally used (CPS functions never return). Since these variables are live in the source program when the function started executing (but they can be reclaimed sooner), the overhead in the live space of the target program cannot exceed the amount of allocation that happens until the next function call, which is proportional to the size of the function body. Hence, the overhead is upper-bounded by the size of the program.

2.2 Linked Closure Representation

In the flat environments in Figure 1b, free variables x, y and z appear multiple times. This increases both the time and the space that the program consumes: when creating environments for nested functions the values of their free variables must be copied from the environment of the immediately enclosing function to the newly constructed environment and, in addition, these environments can be live in the heap, simultaneously holding multiple copies of the same value.

The linked closure representation attempts to avoid this by introducing pointers from the environments of nested functions to the environments of the immediately enclosing functions.

When function k is nested within function h, the linked closure for k contains (locally) only the variables free in k but *not* in h, and points to h's environment for the remaining free variables of k.

Although this representation might save time and space through sharing of free variables across function environments, it is not safe for space. In particular, any closure at some nesting depth can access the closure environments of the all the enclosing functions. This extends lifetime of variables across function calls, and can introduce memory leaks.

Figure 2 shows an example adapted from Appel [1992]. The function *double n* creates a list that repeats 0, n times, then it returns a function that computes the length of the list and that, in turn, returns another function that adds the computed length to n. We expect the space complexity of the program to be $O(M)$: every call to *double* requires $O(M)$ space and in the final result we store M closures, each of them taking up constant space. With linked closures, however, each closure that is created for g will maintain a pointer to the closure environment of f that in turn points to a list of length 1 to M, that cannot be reclaimed by the garbage collector, even though it is not needed. Since the final list keeps M closures, the space required for the execution of the program is $O(M^2)$.

```
let double (n : ℤ) : unit → unit → ℤ =
  let l = repeat 0 n in
  let f () =
    let m = length l in
    let g () = m + n in
    g
  in f

let app (n : ℤ) : list (unit → ℤ) =
  if n = 0 then []
  else double n () :: app (n − 1)

let res = app M
```

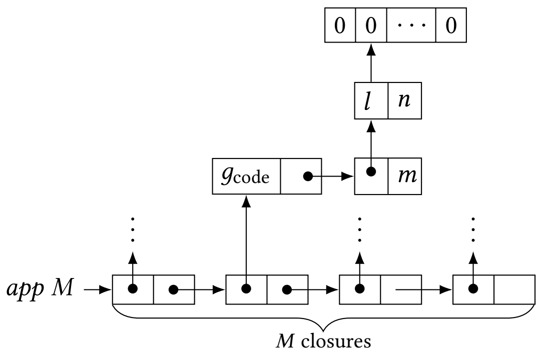


Fig. 2. Linked closures are not safe for space. Each g closure contains (indirectly) a *different* long list l, while a flat closure for g would contain only the two integer values of m and n.

The Javascript V8 engine uses linked environments. Furthermore, it attempts to save time and space by sharing environments between all closures that are defined in the same scope. Environments are created eagerly upon entry to a scope. Although this may appear reasonable as it reduces closure creation time, if the free variables of different functions have different (future) lifetimes, it is not safe for space. As in the previous case, variable lifetimes may extend past the lifetime of functions they are originally used, and lead to asymptotically worse space consumption [Egorov 2012; Glasser 2013].

Therefore: since unsafe-for-space closures can worsen the memory usage of a program by much more than a constant factor, it is important to use safe-for-space closure conversion.

2.3 The Main Theorem

The top-level corollary that we wish to establish states: Let e be a closed program in our compiler's CPS intermediate language. Let $\bar{e}$, in the same language, be the output of the closure-conversion phase. Suppose e evaluates to value v_1 and final heap H_1, using time c_1 and space m_1. Then $\bar{e}$ evaluates to value v_2 with heap H_2 using time c_2 and space m_2, such that: (v_1, H_1) relates to (v_2, H_2),

$c_1 \leq c_2 \leq K_{\mathtt{time}} * c_1$, and $m_2 \leq m_1 + \mathtt{cost}^{\mathtt{space}}_{\mathtt{exp}}(e)$; where $K_{\mathtt{time}}$ is a small constant and $\mathtt{cost}^{\mathtt{space}}_{\mathtt{exp}}(e)$ a factor proportional to the size of the source program e. The additive factor in the space bound accounts for the amount of allocation that happens during the evaluation of the expression e (i.e. until the next function call happens or the program returns), that is the maximum that the space consumption of the target can diverge from the idealized space consumption of the source program.

In addition, a second corollary applies to programs e that diverge: $\bar{e}$ diverges, but its space consumption is guaranteed to remain within bounds.

3 LANGUAGE AND MEMORY MODEL

Our compiler CPS-converts into a continuation-passing style intermediate representation, and the back-end uses heap-allocated closures, including continuation-closures, and no runtime stack. This greatly simplifies the interface between the compiler and garbage collector: there is no need to specify how to find roots in the stack. Such specifications can be enormously complicated [Diwan et al. 1992]; furthermore, no C compiler (and certainly no verified C compiler) supports a method to keep track of root-pointers in the stack. McCreight et al. [McCreight et al. 2010] show how to make a source-to-source transformation in C to keep track of roots, but the overhead of this technique is likely to be larger than the cost of heap closures measured by Shao and Appel. One might think heap-allocated closures impose a significant performance cost, but measurements show that (in a system with efficient generational garbage collection) they are just as efficient as stacks [Appel and Shao 1996]

We begin our formalization with the definition of the untyped CPS lambda calculus (λ_{CPS}) that we use as an intermediate representation in our compiler and on which we apply the closure conversion transformation. We also formalize the heap model used by the semantics of our language that we describe in section 5.

3.1 Syntax

$$
\begin{array}{lccrcll}
(\mathit{Variables}) & x, y & \in & \mathtt{Var} & & & \\
(\mathit{Constructors}) & \mathtt{C} & \in & \mathtt{Constr} & & & \\
(\mathit{Expressions}) & e & \in & \mathtt{Exp} & ::= & \mathtt{let}\ x\ =\ \mathtt{C}(\vec{y})\ \mathtt{in}\ e & \text{Construction} \\
 & & & & \mid & \mathtt{let}\ x\ =\ y.i\ \mathtt{in}\ e & \text{Projection} \\
 & & & & \mid & \mathtt{case}\ y\ \mathtt{of}\ \{\mathtt{C}_i\ \rightarrow\ e_i\}_{i \in I} & \text{Case} \\
 & & & & \mid & \mathtt{let\ rec}\ f\ \vec{x}\ =\ e_1\ \mathtt{in}\ e_2 & \text{Function def.} \\
 & & & & \mid & f\ \vec{x} & \text{Continuation call} \\
 & & & & \mid & \mathtt{halt}(x) & \text{Halt} \\
(\mathit{Contexts}) & \mathcal{E} & \in & \mathtt{Ctx} & ::= & [\cdot] \mid \mathtt{let}\ x\ =\ \mathtt{C}(\vec{y})\ \mathtt{in}\ \mathcal{E} \mid \mathtt{let}\ x\ =\ y.i\ \mathtt{in}\ \mathcal{E} & \\
 & & & & \mid & \mathtt{let\ rec}\ f\ \vec{x}\ =\ e\ \mathtt{in}\ \mathcal{E} & \\
(\mathit{Locations}) & l & \in & \mathtt{Loc} & & & \\
(\mathit{Values}) & v & \in & \mathtt{Val} & ::= & l \mid \mathtt{let\ rec}\ f\ =\ e & \\
(\mathit{Environments}) & \sigma & \in & \mathtt{Env} & = & \mathtt{Var} \rightharpoonup \mathtt{Val} & \\
(\mathit{Blocks}) & b & \in & \mathtt{Block} & ::= & \mathtt{C}(\vec{v}) \mid \mathtt{Clo}(v_1, v_2) \mid \mathtt{Env}(\sigma) & \\
(\mathit{Heaps}) & h & \in & \mathtt{Heap} & = & \mathtt{Loc} \rightharpoonup \mathtt{Block} &
\end{array}
$$

Fig. 3. Syntax and memory model of λ_{CPS}

Expressions. Our intermediate language (fig. 3) is a continuation-passing style lambda calculus with mutually[3] recursive functions, constructors, projections and pattern matching. We use explicit continuation-passing style; the results of any operation are let-bound in their continuations. Source-language calls and source-language returns both become function applications in the λ_{CPS}, that is, leaves of the abstract syntax tree. Constructors are applied to a (possibly empty) list of variables, and projectors project the ith field of a variable. As usual in CPS, we have a `halt` construct that denotes the continuation that will return the result of the evaluation to the top-level.

Our pattern-matching construct might seem slightly odd: we discriminate the value based on the tag of its constructor but without binding the constructor arguments. It is the responsibility of the pattern expression to perform the appropriate projections to access the constructor arguments. In typed source languages, it is common to combine these into a `match` construct that is more easily typechecked; but our untyped CPS has simpler combinational primitives for ease of analysis, optimization, and code generation.

We also define evaluation contexts $\mathcal{E}$ for the expressions of the calculus. We use contexts to describe new code introduced by closure conversion, so we include only the relevant constructors of the language.

Values and Blocks. Values in our language are either pointers to heap blocks or function pointers. We model function pointers as the actual code of the function: this space taken up by the code of the program is statically known and does not change during the execution. We do not account for executable code *in memory*.

A heap block represents either a constructed value, a closure, or an environment. A block representing a constructor contains the constructor tag followed by pointers to the constructor arguments. Before closure conversion, closures are represented as a special type of block consisting of a pair of two values, with the first one expected to be a code pointer and the second one an environment. After closure conversion, closures and their environments will be explicitly constructed by the program and will be represented as constructed values, therefore the target code will only use blocks that represent constructed values. Finally, a heap is represented as a partial map from locations to blocks.

Heap Implementation. In our formal development, our definitions are parameterized (using Coq's module system) by a concrete heap implementation that satisfies an abstract interface. To prove the realizability of our abstract heap model, we provide a concrete implementation of heaps that realizes the abstract interface.

We move on to some definitions that will be useful later for the formalization of the source and target cost models and the garbage collector.

Free Locations. The set of locations that appear in a value can be either a singleton or an empty set:

$$\mathsf{FL}_{\mathtt{Val}}(l) = \{l\} \qquad \mathsf{FL}_{\mathtt{Val}}(\mathtt{let\,rec}\ f\ \vec{x}\ =\ e) = \emptyset$$

We can then define the locations that appear free in an environment in a subset S of its domain, where $\sigma[S]$ is the image of S under σ.

$$\mathsf{FL}_{\mathsf{Env}}(\sigma)[S] = \bigcup_{v \in \sigma[S]} (\mathsf{FL}_{\mathtt{Val}}(v))$$

We simply write $\mathsf{FL}_{\mathsf{Env}}(\sigma)$ to denote the locations of an environment on its whole domain. This is a useful definition: the set $\mathsf{FL}_{\mathsf{Env}}(\sigma)[\mathsf{FV}(e)]$ will be the root set of garbage collection.

[3]We omit mutual recursion in the paper presentation of the calculus.

Using the above definition we can give a definition for the locations that appear free inside a memory block.

$$\begin{array}{rclcrcl} \mathsf{FL}_{\mathsf{Block}}(\mathsf{C}(\vec{v})) & = & \bigcup_{v \in \vec{v}}(\mathsf{FL}_{\mathsf{Val}}(v)) & & \mathsf{FL}_{\mathsf{Block}}(\mathsf{Env}(\sigma)) & = & \mathsf{FL}_{\mathsf{Env}}(\sigma) \\ \mathsf{FL}_{\mathsf{Block}}(\mathsf{Clo}(v_1, v_2)) & = & \mathsf{FL}_{\mathsf{Val}}(v_1) \cup \mathsf{FL}_{\mathsf{Val}}(v_2) & & & & \end{array}$$

Heap Reachability. Given a set of locations S and a heap H, we define the set of successor locations of S as follows.

$$\mathsf{Post}(H)[S] = \bigcup_{b \in H[S]} (\mathsf{FL}_{\mathsf{Block}}(b))$$

Intuitively, this defines the set of locations that we can reach through H in one dereferencing step from the set S. The reachable locations from a root set S of pointers can be defined as the least fixed point of the Post operator.

$$\mathcal{R}(H)[S] = \bigcup_{n \in \mathbb{N}} ((\mathsf{Post}(H))^n [S])$$

Heap Size. To give a formal account for the space consumption of a program, we shall first define the size of a heap. We consider values to be word-sized, as they are either heap or function pointers. First, we define the size of a heap-allocated block as the number of words that it takes up in memory. Blocks that represent constructed values have size equal to a word, that is used to represent the constructor tag, plus the number of the arguments, which are all word-sized values. The size of a block that represents a closure has a constant size equal to three words that are taken up by the tag and the two values that follow it. Lastly, environments have size equal to a word (for the tag) plus the number of bindings in the environment.

$$\mathsf{size}(\mathsf{C}(\vec{v})) = 1 + \mathsf{length}(\vec{v}) \qquad \mathsf{size}(\mathsf{Clo}(v_1, v_2)) = 3 \qquad \mathsf{size}(\mathsf{Env}(\sigma)) = 1 + |\sigma|$$

We then define the size of the heap as the sum of the sizes of the allocated blocks.

$$\mathsf{size}(H) = \sum_{l \in \mathsf{dom}(H)} \mathsf{size}(H(l))$$

It is also useful to define the size of the reachable portion of the heap from a root set S, by restricting the sum to only those blocks that are both in the domain of the heap and reachable from the root set.

$$\mathsf{size}_{\mathcal{R}}(H)[S] = \sum_{l \in \mathsf{dom}(H) \cap \mathcal{R}(H)[S]} \mathsf{size}(H(l))$$

4 HEAP ISOMORPHISM

To specify garbage collection we first formalize a notion of heap isomorphism. After garbage collection the resulting heap will not necessarily be a subheap of the initial heap. Garbage collection algorithms may copy the contents of a location into another to compact the allocated space. Hence, the specification of garbage collection must describe that the reachable portions of the heap before and after garbage collection are equal up to an injective renaming of locations. Injectivity ensures that the same amount of sharing happens before and after garbage collection.[4]

We start by defining a relation on pairs of values and heaps that holds when two values represent the same data structure in their corresponding heap. We define this relation simultaneously with the corresponding relations for environments and heap blocks. These definitions recursively look up in the heap following the pointers referenced by heap blocks to check if the data structures that

[4]Conventional garbage collectors are injective; hash-consing collectors [Appel and Gonçalves 1993] are not.

are represented are the same. To ensure well-foundedness of our definitions in the presence of heap cycles[5], we index them by a natural number indicating the maximum lookup depth in the heap.

Value equivalence[6] is denoted $(v_1, H_1) \approx^n_\beta (v_2, H_2)$ where n is the lookup index and β a location renaming, represented as a total function on locations. Two values are equivalent if they are either both equivalent locations or equivalent function pointers. Two locations are equivalent if a.) they agree on the location renaming and b.) they are either both undefined or both point to equivalent blocks (denoted $(b_1, H_1) \sim^i_\beta (b_2, H_2)$ and its definition is given below). Two function pointers are equivalent if they syntactically represent the same function.

$$\begin{array}{rcll}
(l_1, H_1) \approx^n_\beta (l_2, H_2) & \overset{\text{def}}{=} & l_2 = \beta(l_1) \wedge & \\
 & & \forall i < n, (b_1, H_1) \sim^i_\beta (b_2, H_2) & \text{if } H_1(l_1) = b_1 \text{ and } H_2(l_2) = b_2 \\
(l_1, H_1) \approx^n_\beta (l_2, H_2) & \overset{\text{def}}{=} & l_2 = \beta(l_1) & \text{if } l_1 \notin \mathsf{dom}(H_1) \text{ and } l_2 \notin \mathsf{dom}(H_2) \\
(v, H_1) \approx^n_\beta (v, H_2) & \overset{\text{def}}{=} & \mathtt{True} & \text{if } v = \mathtt{let\ rec}\ f\ \vec{x}\ =\ e \\
(v_1, H_1) \approx^n_\beta (v_2, H_2) & \overset{\text{def}}{=} & \mathtt{False} & \text{otherwise}
\end{array}$$

We define environment equivalence and then use this to define equivalence on blocks representing closure environments. Two environments are equivalent in a set S of free variables, denoted $S \vdash (\sigma_1, H_1) \dot{\approx}^n_\beta (\sigma_2, H_2)$, if for any variable in the set, both environments map the variable to values that are equivalent in the given heaps, or both mappings are undefined.

$$\begin{array}{c}
S \vdash (\sigma_1, H_1) \dot{\approx}^n_\beta (\sigma_2, H_2) \overset{\text{def}}{=} \forall x \in S,\ (\exists v_1\ v_2,\ \sigma_1(x) = v_1\ \wedge\ \sigma_2(x) = v_2\ \wedge\ (v_1, H_1) \approx^n_\beta (v_2, H_2)) \vee \\
(x \notin \mathsf{dom}(\sigma_1)\ \wedge\ x \notin \mathsf{dom}(\sigma_2))
\end{array}$$

When the set S is the full set of variables we simply write $(\sigma_1, H_1) \dot{\approx}^n_\beta (\sigma_2, H_2)$.

Using the above definitions we can define block equivalence. Two heap blocks are equivalent in their corresponding heaps if they are both constructed values with the same outermost constructor and arguments that are pairwise equivalent values, or if they are both closures whose arguments are pairwise equivalent values, or if they are both equivalent environments.

$$\begin{array}{rcll}
(\mathsf{C}(v_1, \dots v_n), H_1) \sim^n_\beta (\mathsf{C}(v'_1, \dots v'_n), H_2) & \overset{\text{def}}{=} & \forall i, (v_i, H_1) \approx^n_\beta (v'_i, H_2) & \\
(\mathsf{Clo}(v_1, v_2), H_1) \sim^n_\beta (\mathsf{Clo}(v'_1, v'_2), H_2) & \overset{\text{def}}{=} & (v_1, H_1) \approx^n_\beta (v'_1, H_2) \wedge & \\
 & & (v_2, H_1) \approx^n_\beta (v'_2, H_2) & \\
(\mathsf{Env}(\sigma_1), H_1) \sim^n_\beta (\mathsf{Env}(\sigma_2), H_2) & \overset{\text{def}}{=} & (\sigma_1, H_1) \dot{\approx}^n_\beta (\sigma_2, H_2) & \\
(b_1, H_1) \sim^n_\beta (b_2, H_2) & \overset{\text{def}}{=} & \mathtt{False} & \text{otherwise}
\end{array}$$

That concludes the (mutually recursive) definition of value equivalence. We can now define value, environment, and block equivalence for any lookup depth simply by quantifying over all lookup indexes. To simplify future definitions, we also require that the renaming is an injective

[5]Our λ_{CPS} does not build circular data structures, but our framework is extensible to (e.g.,) mutable references.

[6]We use the term "equivalence" only nominally for these definitions. Only if we existentially quantify the location renaming are these relations equivalences. However, we want to keep the location renaming transparent in our definitions in order to use it in the semantics.

function in the set of reachable locations.

$$\begin{aligned}
(l_1, H_1) \approx_\beta (l_2, H_2) &\overset{\text{def}}{=} \forall\, n,\ (l_1, H_1) \approx^n_\beta (l_2, H_2)\ \wedge\ \mathtt{inj}_{\mathcal{R}(H_1)[\{l_1\}]}(\beta) \\
S \vdash (\sigma_1, H_1) \dot{\approx}_\beta (\sigma_2, H_2) &\overset{\text{def}}{=} \forall\, n,\ S \vdash (\sigma_1, H_1) \dot{\approx}^n_\beta (\sigma_2, H_2)\ \wedge\ \mathtt{inj}_{\mathcal{R}(H_1)[\mathsf{FL}_{\mathsf{Env}}(\sigma_1)[S]]}(\beta) \\
(b_1, H_1) \sim_\beta (b_2, H_2) &\overset{\text{def}}{=} \forall\, n,\ (b_1, H_1) \sim^n_\beta (b_2, H_2)\ \wedge\ \mathtt{inj}_{\mathcal{R}(H_1)[\mathsf{FL}_{\mathsf{Block}}(b_1)]}(\beta)
\end{aligned}$$

Given a root set of locations and a renaming of locations, two heaps are isomorphic w.r.t. this renaming if any location in the set is equivalent with its renaming in the second heap.

$$S \vdash H_1 \dot{\sim}_\beta H_2 \overset{\text{def}}{=} \forall\, l \in S,\ (l, H_1) \approx_\beta (\beta(l), H_2)$$

In the following section we will use this definition to give a specification for garbage collection.

5 PROFILING SEMANTICS

In this section we formalize the source and target level semantics of λ_{CPS}. We use environment-based, big-step operational semantics for both the source and target. Although the source and target languages of closure conversion are the same, that is not true for the semantics. Before closure conversion, the semantics need to implicitly construct a closure for each function definition by capturing the relevant part of the environment, and storing it in the heap in a special block type for closures. After closure conversion, the code explicitly constructs environments as ordinary constructed values, and therefore the target semantics only needs to store the pointer to the closed function. Our source and target semantics profile the time and space needed for the execution of the program. Before getting into the details of our definitions, we discuss informally how we measure time and space.

Time. We use fuel-based semantics for both the source and the target: the computation times out if there are not enough units of time available. At each execution step the virtual clock winds down by the cost associated with the language constructor that is being evaluated, which is proportional to its size. The execution time of a program is the smallest fuel value that does not time out.

Space. Our source space-cost semantics measures the size of the reachable heap at every evaluation step, and keeps track of the maximum value. That is, a program's space cost is the maximum reachable heap space during its execution. Conceptually, this is the space consumption of an ideal garbage collector strategy that collects all the unreachable pointers after every execution step.

In principle, we can use the same profiling strategy in the target in order to prove that closure conversion is safe for space. However, we opt for a different strategy in the target that models more accurately the runtime of our target. In particular, we keep track of the maximum size of the actual heap (instead of the reachable heap) but we run a garbage collector upon every function entry. Intuitively, this preserves the idealized measurement of space usage because only a bounded amount of allocation can occur between function calls in CPS. Functions in continuation-passing style are trees of control-flow (no loops) whose leaves are function calls, therefore the amount of heap allocation between function calls is bounded by the path-length of the function. Therefore we overapproximate the truly live data (compared to measuring at every allocation) by an additive amount proportional at worst to the size of the program. By using this profiling strategy in the target, not only we prove that closure conversion is safe-for-space but also that garbage collecting the heap upon every function entry is a safe-for-space garbage-collection strategy. Although this strategy is still idealized with respect to what our real garbage collector is doing, it allows us to carry out a part of the refinement proof between the ideal model and our actual garbage collection implementation "for free".

Theorem 5.1 (not proved in Coq). *If a program runs in our target semantics with time t and space A, then there exists a simple garbage collector with which the program can run in time linear in t and space linear in A.*

Proof. Use a simple two-semispace copying collector [Appel 1992, Ch. 16] in memory $M = 4A$. Each semispace is size $2A$, and when it is full, the collector copies it to the other semispace using Cheney's algorithm. Construct an alternate profiling semantics for the target language, which runs the garbage collector at each function call if the size of the heap is $\geq M/4$.

In the alternate profiling semantics, one garbage collection takes cA time, for some constant c. The program can run for at least A instructions (it takes at least one instruction to initialize each word of a newly allocated record) before the next collection. Therefore, the total time to run the program in our alternate semantics is bounded by $(c + 1)t$, and the memory use of our alternate semantics is $4A$.

Finally, there is a simple simulation relation between our "ideal-garbage-collector" target semantics (formalized below) and our "real-garbage-collector" target semantics (which we do not formalize here). The simulation permits unreachable data to be absent from the ideal heap, but present (not yet collected) in the real heap. □

Our specification permits the use of other (more efficient) garbage collection algorithms, such as generational collection. For any such algorithm, one can formulate a "real-garbage-collector" target semantics, prove bounds on the space and time usage of the program, and prove a simulation between the ideal and the real.

5.1 Formal Garbage Collection Model

We give a relational specification for ideal garbage collection. Given a root set S we specify when one heap is the collection of another for a given location renaming β.

$$\mathsf{GC}_S(H_1, H_2, \beta) \quad \overset{\text{def}}{=} \quad S \vdash H_1 \mathrel{\dot{\sim}_\beta} H_2 \ \land\ \mathsf{dom}(H_2) \subseteq \mathcal{R}(H_2)[\beta(S)]$$

The two heaps must be isomorphic (up to the given renaming) in the set of the locations reachable from the root set in the initial heap (this also implies that the renaming is injective). To ensure that all the unreachable locations are collected, we require that the domain of the collected heap is a subset of the locations reachable from the root set in the collected heap. We apply the renaming in the initial root set, to find the root set of the collected heap.

To justify our specification of garbage collection we implement in Coq a simple garbage collection algorithm for our concrete heap implementation and we prove that it satisfies the above specification. Our algorithm simply computes the set of reachable locations from the root set and restricts the domain of the heap to this set. This algorithm yields the identity renaming, since locations are not moved.

5.2 Operational Semantics

Our big-step operational semantics judgment, written $H; \sigma; e \Downarrow_l^{(i,m)} r$, with $l \in \{\mathtt{src}, \mathtt{trg}\}$ for the source and target semantics respectively, states that a configuration ($\mathsf{Conf} \in \mathsf{Heap} \times \mathsf{Env} \times \mathsf{Exp}$) evaluates to a result r. It is indexed by two metrics: a virtual clock i indicating the available execution steps (decreasing as the program executes), and m indicating the consumed space during execution. The result can be either a pair of a value and a heap or an out-of-time exception, written OOT, that indicates that the program does not terminate within i steps.

We adopt a fuel-based execution cost model to reason about nonterminating programs and their heap consumption. A program diverges if for all values of the clock it yields an out-of-time exception.

$$H; \sigma; e \Uparrow_{l}^{m} \quad \overset{\text{def}}{=} \quad \forall\, i,\ \exists\, m',\ H; \sigma; e \Downarrow^{(i,m')} \texttt{OOT} \ \land\ m' \leq m$$

The definition is annotated with an upper bound for memory consumption of the diverging program, which might be infinite and hence it belongs to the set $\mathbb{N} \cup \{\infty\}$.

Source Semantics. The evaluation rules for the source semantics are shown in fig. 4. In inductive cases, the memory consumption is the maximum of the memory consumption of the recursive call and the size of reachable heap before evaluating the outermost constructor. In base cases (rules Base and OOT) the memory consumption is the size of reachable heap. At every evaluation step we decrease the virtual clock by the cost of each rule, which is given by the function $\texttt{cost}(e)$. If at any point the remaining units of computation are less that the units required to evaluate the outermost constructor the programs terminates with an out-of-time exception.

Most rules are straightforward; here we review the slightly more complicated rules for function definition and application. The rule for function definition (rule Fun) first allocates the environment of the closure, that is the current environment restricted to these variables that appear free in the function body. It then allocates a closure and it binds the function name to the closure pointer in the environment, before evaluating the continuation of the expression. The cost of evaluating a function declaration is a unit of time plus the number of the free variables of the function, to account for the implicit closure environment creation. The rule for function application (rule App) looks up the value of the applied function in the environment which should be a pointer to a closure in the heap. It dereferences the environment pointer of the closure in order to find the closure environment, which extends by binding the formal parameters to the values of the actual parameters and the name of the function to the closure pair. It then evaluates the body of the function in the extended closure environment.

Target Semantics. The semantics for evaluating the code after closure conversion need not construct the closure pair and environment, as this is done explicitly by the code. Therefore the application and function definition cases need to be handled differently in the target semantics. Furthermore, we change the way space profiling works: we profile the actual size of the heap, and we invoke the garbage collector upon function entry. Since heap grows monotonically except for when garbage collection is invoked, it suffices to measure the space in base cases and just before garbage collection and keep track of the maximum space measured during execution. As in the source semantics, evaluating each constructor incurs a cost. In this case the cost associated to the function definition will be constant since the function is closed and the semantics will not construct the closure environment.

Selected evaluation rules are shown in fig. 5. In the function definition case (rule Fun$_{\text{CC}}$), the environment is extended with the function name bound to function pointer before evaluating the continuation. In the application case (rule App$_{\text{CC}}$), the values of the actual parameters and the value of the function that is being looked up in the environment, with the latter expected to be a function pointer. A new environment is constructed by binding the names of the formal parameters to the values of the actual parameters and the function name to the function pointer. Before continuing to the evaluation of the function body, the heap is garbage collected using as roots the locations in environment bindings of the variables of the function.

We define the interpretation of an evaluation context in some given heap and environment, formalized as a judgment of the form $H_1; \sigma_1; \mathcal{E} \blacktriangleright_{l}^{c} H_2; \sigma_2$, with $l \in \{\texttt{src}, \texttt{trg}\}$. The judgment asserts that the evaluation context $\mathcal{E}$ is interpreted in heap H_1 and environment σ_1, yielding a new heap H_2 and environment σ_2, using c units of time. The rules of this judgment follow closely the rules of the evaluation judgment; we do not show them here.

$$\frac{\begin{array}{c}\sigma(\vec{y}) = \vec{v} \qquad \texttt{alloc}(\texttt{C}(\vec{v}), H_1) = (l, H_2) \\ H_2; \sigma[x \mapsto l]; e \Downarrow_{\textsf{src}}^{(i-c,m')} r \qquad c \leq i \qquad c = \texttt{cost}(\texttt{let}\ x\ =\ \texttt{C}(\vec{y})\ \texttt{in}\ e) \\ m = \max(m', \texttt{size}_{\mathcal{R}}(H_1)[\textsf{FL}_{\textsf{Env}}(\sigma)[\textsf{FV}(\texttt{let}\ x\ =\ \texttt{C}(\vec{y})\ \texttt{in}\ e)]])\end{array}}{H_1; \sigma; \texttt{let}\ x\ =\ \texttt{C}(\vec{y})\ \texttt{in}\ e \Downarrow_{\textsf{src}}^{(i,m)} r}\ \textsc{Constr}$$

$$\frac{\begin{array}{c}\sigma(y) = l \qquad H(l) = \texttt{C}(v_1, \ldots, v_j, \ldots, v_n) \qquad H; \sigma[x \mapsto v_j]; e \Downarrow_{\textsf{src}}^{(i-c,m')} r \qquad c \leq i \\ c = \texttt{cost}(\texttt{let}\ x\ =\ y.j\ \texttt{in}\ e) \qquad m = \max(m', \texttt{size}_{\mathcal{R}}(H_1)[\textsf{FL}_{\textsf{Env}}(\sigma)[\textsf{FV}(\texttt{let}\ x\ =\ y.j\ \texttt{in}\ e)]])\end{array}}{H; \sigma; \texttt{let}\ x\ =\ y.j\ \texttt{in}\ e \Downarrow_{\textsf{src}}^{(i,m)} r}\ \textsc{Proj}$$

$$\frac{\begin{array}{c}\sigma(x) = l \\ H(l) = \texttt{C}_i(\vec{v}) \qquad H; \sigma; e_i \Downarrow_{\textsf{src}}^{(i-c,m')} r \qquad c \leq i \qquad c = \texttt{cost}(\texttt{case}\ x\ \texttt{of}\ \{\texttt{C}_i\ \rightarrow\ e_i\}_{i \in I}) \\ m = \max(m', \texttt{size}_{\mathcal{R}}(H_1)[\textsf{FL}_{\textsf{Env}}(\sigma)[\textsf{FV}(\texttt{case}\ x\ \texttt{of}\ \{\texttt{C}_i\ \rightarrow\ e_i\}_{i \in I})]])\end{array}}{H; \sigma; \texttt{case}\ x\ \texttt{of}\ \{\texttt{C}_i\ \rightarrow\ e_i\}_{i \in I} \Downarrow_{\textsf{src}}^{(i,m)} r}\ \textsc{Case}$$

$$\frac{\begin{array}{c}\texttt{alloc}(\textsf{Env}(\sigma|_{\textsf{FV}(\texttt{let rec}\ f\ \vec{x}\ =\ e_1)}), H_1) = (l_{env}, H_2) \\ \texttt{alloc}(\texttt{Clo}(\texttt{let rec}\ f\ \vec{x}\ =\ e_1, l_{env}), H_2) = (l_f, H_3) \\ H_3; \sigma[f \mapsto l_f]; e_2 \Downarrow_{\textsf{src}}^{(i-c,m')} r \qquad c \leq i \qquad c = \texttt{cost}(\texttt{let rec}\ f\ \vec{x}\ =\ e_1\ \texttt{in}\ e_2) \\ m = \max(m', \texttt{size}_{\mathcal{R}}(H_1)[\textsf{FL}_{\textsf{Env}}(\sigma)[\textsf{FV}(\texttt{let rec}\ f\ \vec{x}\ =\ e_1\ \texttt{in}\ e_2)]])\end{array}}{H_1; \sigma; \texttt{let rec}\ f\ \vec{x}\ =\ e_1\ \texttt{in}\ e_2 \Downarrow_{\textsf{src}}^{(i,m)} r}\ \textsc{Fun}$$

$$\frac{\begin{array}{c}\sigma(\vec{x}) = \vec{v} \qquad \sigma(f) = l_f \qquad H_1(l_f) = \texttt{Clo}(\texttt{let rec}\ g\ \vec{x}\ =\ e_1, l_{env}) \\ H_1(l_{env}) = \textsf{Env}(\sigma_f) \qquad H_1; \sigma_f[\vec{x} \mapsto \vec{v}][g \mapsto l_f]; e_1 \Downarrow_{\textsf{src}}^{(i-c,m')} r \\ c \leq i \qquad c = \texttt{cost}(f\ \vec{x}) \qquad m = \max(m', \texttt{size}_{\mathcal{R}}(H_1)[\textsf{FL}_{\textsf{Env}}(\sigma)[\textsf{FV}(f\ \vec{x})]])\end{array}}{H_1; \sigma; f\ \vec{x} \Downarrow_{\textsf{src}}^{(i,m)} r}\ \textsc{App}$$

$$\frac{\sigma(x) = v \qquad\qquad \texttt{cost}(\texttt{halt}(x)) \leq i}{H; \sigma; \texttt{halt}(x) \Downarrow_{\textsf{src}}^{(i, \texttt{size}_{\mathcal{R}}(H_1)[\textsf{FL}_{\textsf{Env}}(\sigma)[\textsf{FV}(\texttt{halt}(x))]])} (v, H)}\ \textsc{Halt}$$

$$\frac{i < \texttt{cost}(e)}{H; \sigma; e \Downarrow_{\textsf{src}}^{(i, \texttt{size}_{\mathcal{R}}(H_1)[\textsf{FL}_{\textsf{Env}}(\sigma)[\textsf{FV}(e)]])} \textsf{OOT}}\ \textsf{OOT}$$

where

$$\begin{array}{rcl@{\qquad}rcl}
\texttt{cost}(\texttt{let}\ x\ =\ \texttt{C}(\vec{y})\ \texttt{in}\ e) & \stackrel{\text{def}}{=} & 1 + |\vec{y}| & \texttt{cost}(\texttt{let}\ x\ =\ y.j\ \texttt{in}\ e) & \stackrel{\text{def}}{=} & 1 \\
\texttt{cost}(\texttt{case}\ x\ \texttt{of}\ \{\texttt{C}_i\ \rightarrow\ e_i\}_{i \in I}) & \stackrel{\text{def}}{=} & 1 & \texttt{cost}(\texttt{halt}(x)) & \stackrel{\text{def}}{=} & 1 \\
\texttt{cost}(\texttt{let rec}\ f\ \vec{x}\ =\ e_1\ \texttt{in}\ e_2) & \stackrel{\text{def}}{=} & 1 + |\textsf{FV}(\texttt{let rec}\ f\ \vec{x}\ =\ e_1)| & \texttt{cost}(f\ \vec{x}) & \stackrel{\text{def}}{=} & 1 + |\vec{x}|
\end{array}$$

Fig. 4. Big-step operational semantics (source)

6 CLOSURE CONVERSION

In this section we give a formal account of the flat closure conversion transformation for our CPS intermediate representation. The closure converted code will explicitly construct closure environment and pairs, and will turn free variable occurrences to environment accesses. Function

$$\frac{\begin{array}{c} H; \sigma[f \mapsto \texttt{let rec}\ f\ \vec{x}\ =\ e_1]; e_2 \Downarrow_{\texttt{trg}}^{(i-c,m)} r \\ c \leq i \qquad c = \texttt{cost}(\texttt{let rec}\ f\ \vec{x}\ =\ e_1\ \texttt{in}\ e_2) \end{array}}{H; \sigma; \texttt{let rec}\ f\ \vec{x}\ =\ e_1\ \texttt{in}\ e_2 \Downarrow_{\texttt{trg}}^{(i,m)} r}\ \textsc{Fun}_{\textsc{cc}} \qquad \frac{\sigma(x) = v \qquad \texttt{cost}(f\ \vec{x}) \leq i}{H; \sigma; \texttt{halt}(x) \Downarrow_{\texttt{trg}}^{(i,\texttt{size}(H_1))} (v, H)}\ \textsc{Halt}$$

$$\frac{\begin{array}{c} \sigma(\vec{x}) = \vec{v} \qquad \sigma(f) = \texttt{let rec}\ g\ \vec{x}\ =\ e \qquad \texttt{GC}_{\texttt{FL}_{\texttt{Env}}([\vec{x} \mapsto \vec{v}][g \mapsto \texttt{let rec}\ g\ \vec{x}\ =\ e])[\texttt{FV}(e)]}(H_1, H_2, \beta) \\ H_2; \beta \circ ([\vec{x} \mapsto \vec{v}][g \mapsto \texttt{let rec}\ g\ \vec{x}\ =\ e]); e \Downarrow_{\texttt{trg}}^{(i-c,m')} r \\ c \leq i \qquad c = \texttt{cost}(f\ \vec{x}) \qquad m = \max(m', \texttt{size}(H_1)) \end{array}}{H_1; \sigma; f\ \vec{x} \Downarrow_{\texttt{trg}}^{(i,m)} r}\ \textsc{App}_{\textsc{cc}}$$

Fig. 5. Big-step operational semantics (target)

calls will be modified so that they destruct the closure pair and pass the closure environment as an argument to the function. Therefore, the post-closure-conversion semantics (that we presented in the previous section) does not need to capture the environment at the time of function definition or look it up in the heap during function calls. Functions are closed and therefore can be evaluated at the environment containing bindings for the formal parameters and the function name. After closure conversion, all function definitions can be hoisted to the top level, and there are only two levels of scope: the global scope and the scope that is local to every function. The resulting flat, closure-converted code can be directly translated to C.

6.1 Closure Conversion

We present closure conversion as a deductive system. The judgment $\Gamma, \Phi \vdash_{(\phi,\gamma)} e \rightsquigarrow \bar{e}$ asserts that $\bar{e}$ is the output of closure conversion of a source program e in a *local* environment Γ, that contains the names of locally bound free variables (i.e., variables declared within the scope of the current function declaration) and a *global* environment Φ, that contains the names of free variables in scopes not local to the current function. The judgment is also indexed by two identifiers ϕ (that maps function names in scope to their closure environment) and γ (the name of the formal parameter that corresponds to the closure environment in the current function). We write $e \rightsquigarrow \bar{e}$ for the closure conversion of top-level programs in which case Γ, Φ are empty and ϕ, γ have dummy values. The global environment Φ is an ordered set, that represents the order in which the values of free variables are stored in the closure environment.

We formalize an auxiliary judgment $\Gamma, \Phi \vdash_{(\phi,\gamma)} \vec{x} \triangleright \mathcal{E}, \Gamma'$ that associates a list of free variables $\vec{x}$ before closure conversion with an evaluation context $\mathcal{E}$, under which the variables should be used in closure converted code, and a possibly updated local environment Γ'. Intuitively, the evaluation context will take care of finding the values of free variables by accessing the environment parameter, and constructing the closure pair when this is needed. We want to project each free variable and construct a closure pair at most once within each local scope, we update the local environment so that the next time the same variable will be used it will not be projected or constructed again. Apart from the local and global environments (Γ and Φ respectively), the judgment has two additional parameters: ϕ a partial map that maps function names in scope to their corresponding closure environments, and γ which is the name of the formal parameter of the environment in the current scope, or a dummy variable if we are at the top-level scope.

The rules of this judgment are presented in fig. 6. If a variable is in the local scope (rule Local), then it can be used as-is. The evaluation context and the local environment do not change. If a pre-closure-conversion variable is in the global environment (rule Global), then we project the

$$\frac{x \in \Gamma \qquad \Gamma, \Phi \vdash_{(\phi,\gamma)} \vec{x} \triangleright \mathcal{E}, \Gamma'}{\Gamma, \Phi \vdash_{(\phi,\gamma)} x :: \vec{x} \triangleright \mathcal{E}, \Gamma'} \textsc{Local}$$

$$\frac{x_i \notin \Gamma \qquad \Phi = x_1, \ldots, x_i, \ldots, x_n \qquad \{x\} \cup \Gamma, \Phi \vdash_{(\phi,\gamma)} \vec{x} \triangleright \mathcal{E}, \Gamma'}{\Gamma, \Phi \vdash_{(\phi,\gamma)} x_i :: \vec{x} \triangleright \texttt{let}\, x_i \;=\; \gamma.\texttt{i in}\, \mathcal{E}, \Gamma'} \textsc{Global}$$

$$\frac{f \notin \Gamma \qquad f \in \texttt{dom}(\phi) \qquad \{f\} \cup \Gamma, \Phi \vdash_{(\phi,\gamma)} \vec{x} \triangleright \mathcal{E}, \Gamma'}{\Gamma, \Phi \vdash_{(\phi,\gamma)} f :: \vec{x} \triangleright \texttt{let}\, f \;=\; \mathsf{C}_{\mathsf{CC}}(f, \phi(f))\, \texttt{in}\, \mathcal{E}, \Gamma'} \textsc{Function} \qquad \frac{}{\Gamma, \Phi \vdash_{(\phi,\gamma)} [] \triangleright [\,\cdot\,], \Gamma} \textsc{Empty}$$

Fig. 6. Free variable judgment

value from the current environment parameter and we assign it to a binder that shadows its name. We add the variable to the local environment, so subsequent uses do not cause it to be projected out again. If a pre-closure-conversion variable is in the domain of the function environment map ϕ (rule Function), then it is a function name that must be packed together with its environment (given by the binding of the map) to construct a closure pair. Again, the local environment is extended accordingly.

We may now formalize the closure-conversion judgment (fig. 7. The rules for constructor, projection, pattern matching and halt are straightforward propagation rules. All they need to do is to handle the free variables correctly using the judgment we formalized above. The function case (rule $\textsc{cc}_{\textsc{Fun}}$), after picking a fresh name for the formal environment parameter of the closure-converted function, closure-converts the body of the function in a local environment that contains only the formal parameters, a global environment that consists of the free variables of the function definition in some particular order and a function map that has a single binding of the currently defined function to its formal environment parameter. The closure-converted code will then construct the closure environment. The function map will be extended with a mapping from the newly defined function to the newly constructed environment, and the continuation will be closure-converted. The application case (rule $\textsc{cc}_{\textsc{App}}$), after handling the free variables g and $\vec{x}$, projects the code pointer and the environment parameter out of the closure and performs the application passing the extra environment argument.

For recursive calls, our closure-conversion transformation will first construct the closure by packing its code with the current environment, then immediately project them out to perform the call. During compilation, these administrative redexes, that do not affect our resource preservation proof, will be eliminated by a proved-correct shrink-reduction transformation [Savary Bélanger and Appel 2017].

In our compiler we have a closure-conversion *program*, a Coq function. Although we could prove it correct directly, we provide this inductive definition as a relational specification for closure conversion. This allows us to reason about closure conversion without worrying about the details of the implementation. Then, we prove that closure-conversion program is correct w.r.t. the relational specification.

7 LOGICAL RELATION

In this section we build the logical relation that we use to prove closure conversion correct and safe for time and space, and we go through some important properties it satisfies. The idea is that along with proving functional correctness we establish the cost bounds on the resources consumed by evaluating the source and target. We achieve this by parameterizing the logical

$$\frac{\Gamma, \Phi \vdash_{(\phi,\gamma)} \vec{y} \triangleright \mathcal{E}, \Gamma' \qquad \{x\} \cup \Gamma', \Phi \vdash_{(\phi,\gamma)} e \rightsquigarrow \bar{e}}{\Gamma, \Phi \vdash_{(\phi,\gamma)} \mathtt{let}\ x\ =\ \mathsf{C}(\vec{y})\ \mathtt{in}\ e \rightsquigarrow \mathcal{E}[\mathtt{let}\ x\ =\ \mathsf{C}(\vec{y})\ \mathtt{in}\ \bar{e}]}\ \textsc{CCConstr}$$

$$\frac{\Gamma, \Phi \vdash_{(\phi,\gamma)} y \triangleright \mathcal{E}, \Gamma' \qquad \{x\} \cup \Gamma', \Phi \vdash_{(\phi,\gamma)} e \rightsquigarrow \bar{e}}{\Gamma, \Phi \vdash_{(\phi,\gamma)} \mathtt{let}\ x\ =\ \vec{y}.i\ \mathtt{in}\ e \rightsquigarrow \mathcal{E}[\mathtt{let}\ x\ =\ y'.i\ \mathtt{in}\ \bar{e}]}\ \textsc{CCProj}$$

$$\frac{\Gamma, \Phi \vdash_{(\phi,\gamma)} x \triangleright \mathcal{E}, \Gamma' \qquad \Gamma', \Phi \vdash_{(\phi,\gamma)} e_i \rightsquigarrow \bar{e}_i}{\Gamma, \Phi \vdash_{(\phi,\gamma)} \mathtt{case}\ x\ \mathtt{of}\ \{\mathsf{C}_i\ \longrightarrow\ e_i\}_{i \in I} \rightsquigarrow \mathcal{E}[\mathtt{case}\ x\ \mathtt{of}\ \{\mathsf{C}_i\ \longrightarrow\ \bar{e}_i\}_{i \in I}]}\ \textsc{CCCase}$$

$$\frac{\Gamma, \Phi \vdash_{(\phi,\gamma)} x \triangleright \mathcal{E}, \Gamma'}{\Gamma, \Phi \vdash_{(\phi,\gamma)} \mathtt{halt}(x) \rightsquigarrow \mathcal{E}[\mathtt{halt}(x)]}\ \textsc{CCHalt}$$

$$\frac{\Gamma, \Phi \vdash_{(\phi,\gamma)} g :: \vec{x} \triangleright \mathcal{E}, \Gamma'}{\Gamma, \Phi \vdash_{(\phi,\gamma)} g\ \vec{x} \rightsquigarrow \mathcal{E}[\mathtt{let}\ g_{\mathrm{code}}\ =\ g.1\ \mathtt{in}\ \mathtt{let}\ g_{\mathrm{env}}\ =\ g.2\ \mathtt{in}\ g_{\mathrm{code}}\ (g_{\mathrm{env}} :: \vec{x'})]}\ \textsc{CCApp}$$

$$\frac{\begin{array}{c}\vec{FV} = \mathsf{FV}(\mathtt{let\ rec}\ f\ \vec{x}\ =\ e_1) \\ \Gamma, \Phi \vdash_{(\phi,\gamma)} \vec{FV} \triangleright \mathcal{E}, \Gamma' \qquad \vec{x}, \vec{FV} \vdash_{([f \mapsto \gamma_f], \gamma_f)} e_1 \rightsquigarrow \bar{e}_1 \qquad \Gamma, \Phi \vdash_{(\phi[f \mapsto f_{\mathrm{env}}], \gamma)} e_2 \rightsquigarrow \bar{e}_2\end{array}}{\Gamma, \Phi \vdash_{(\phi,\gamma)} \mathtt{let\ rec}\ f\ \vec{x}\ =\ e_1\ \mathtt{in}\ e_2 \rightsquigarrow \mathcal{E}[\mathtt{let\ rec}\ f\ \gamma_f :: \vec{x}\ =\ \bar{e}_1\ \mathtt{in\ let}\ f_{\mathrm{env}} = \mathsf{C}_{env}(\vec{FV})\ \mathtt{in}\ \bar{e}_2]}\ \textsc{CCFun}$$

Fig. 7. Closure conversion

relation with a pre- and a postcondition. The precondition is imposed on the initial configurations, and the postcondition is guaranteed to hold on the results of the evaluation. Our logical relation construction is nonstandard in the following ways:

- It is parameterized by two pairs of pre- and postconditions. The doubling of pre- and postconditions allows us to prove monotonicity of the logical relation with respect to these and provide weakening and strengthening rules important for compositional reasoning.
- For a certain class of pre- and postconditions, our reasoning applies to diverging sources as well: we show that divergence is preserved and the memory bound holds. We explain how the combination of fuel-based semantics and time resource bounds enable us to extend the reasoning to nonterminating programs.
- Our logical relation is doubly indexed by a step-index that bounds the depth of recursion and it is crucial for the well-foundedness of our definition, and a heap lookup index that bounds the look up depth in the heap. Although one index could (seemingly) have served both purposes, we explain why it is important to decouple the two, and quantify over all lookup indexes in the fundamental theorem of the logical relation.
- Our logical relation satisfies Kripke monotonicity, in that it respects heap isomorphism: when two configurations are related, they are related for all pairs of configurations that are isomorphic to the original configurations. We achieve this by quantifying over all pairs of isomorphic heap-environment pairs. This allows us to maintain relatedness of the evaluation environments through the execution, regardless of how irrelevant portions of the heap may change.

7.1 Configuration Relation: A Failed Attempt

We wish to define a logical relation that allows us to relate the costs of the source and target programs along with proving semantic preservation. Setting the value relation aside for now, let us try to define the configuration relation, that relates the execution of a source and a target configuration.

Following the standard pattern in step-indexed logical relations, we start by stating that for a given step index when the source evaluates to a result in less steps than the step index, the target program also evaluates to a result that is related with the result of the source with the value relation (denoted with $\mathcal{V}$). Additionally, we parameterize the relation with a *resource precondition* that is imposed on the initial configurations, and a resource postcondition that holds on the two pairs of the time and space consumption. The resource bounds can be program-dependent, hence we add the source configuration as a parameter to the resource postcondition, that is $\mathtt{Post} \in \mathtt{Conf} \to \mathtt{Relation}(nat \times nat)$.

The definition of the logical relation looks like,

$$\begin{aligned}
&\mathcal{C}^k\ \{P\}\ (H_1, \sigma_1, e_1)\ (H_2, \sigma_2, e_2)\ \{Q\} \overset{\text{def}}{=} \\
&\quad \forall\, r_1\ c_1\ m_1,\ P\ (H_1, \sigma_1, e_1)\ (H_2, \sigma_2, e_2)\ \to\ c_1 \le k\ \to\ H_1; \sigma_1; e_1 \Downarrow_{\mathtt{src}}^{(c_1, m_1)} r_1\ \to \\
&\quad\quad \exists\, \sigma_2\ c_2\ m_2,\ H_2; \sigma_2; e_2 \Downarrow_{\mathtt{trg}}^{(c_2, m_2)} r_2\ \wedge\ Q\ (H_1, \sigma_1, e_1)\ (c_1, m_1)\ (c_2,\ m_2)\ \wedge\ \mathcal{V}^{k-c_1}\ \{P\}\ r_1\ r_2\ \{Q\}
\end{aligned}$$

where $\mathcal{C}$ denotes the configuration relation, k is the step index, P and Q the pre- and postcondition, and (H_1, σ_1, e_1) and (H_2, σ_2, e_2) the source and target configurations.

We would have been satisfied with this definition if we could prove that the logical relation inhabits it, as it would entail what we wish to establish for the transformation. But technical complications prevent us from using this definition directly.

Compositional Reasoning. We want to prove compatibility lemmas that we can use in our closure-conversion proof to establish that each step of the transformation yields related programs. For instance, to prove the projection case we would need a compatibility lemma that roughly states,

Lemma 7.1. *Assume that*

- $\mathcal{G}^k\ \{y\} \vdash \{P\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q\}$
- $\forall\, v_1\ v_2, \mathcal{V}^k\ \{P\}\ (v_1, H_1)\ (v_2, H_2)\ \{Q\} \Rightarrow$
  $\mathcal{C}^k\ \{P\}\ (H_1, \sigma_1[x \mapsto v_2], e_1)\ (H_2, \sigma_2[x \mapsto v_2], e_2)\ \{Q\}$

Then $\mathcal{C}^k\ \{P\}\ (H_1, \sigma_1, \mathtt{let}\ x\ =\ y.j\ \mathtt{in}\ e_1)\ (H_2, \sigma_2, \mathtt{let}\ x\ =\ y.j\ \mathtt{in}\ e_2)\ \{Q\}$

where $\mathcal{G}^k\ S \vdash \{P\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q\}$ is the environment relation that asserts that the source and target environments map variables in S to related values.

The above variant of the lemma is problematic because it forces us to use the same pre- and postconditions before and after the evaluation of the outer constructor. This happens because the same pre- and postcondition that are enforced for the current configuration should hold also for future execution of whole functions. Looking back at the closure-conversion rules, before evaluating the right expression we should evaluate the context that causes the projection of the free variables which, if not empty, consumes some units of time. Therefore, after evaluating this context, the initial postcondition will no longer hold. The original postcondition should be reestablished after the evaluation of the two projection constructors, allowing us to apply the induction hypothesis. To support compositional reasoning we should allow pre- and postconditions to vary in these rules. As we explain below, we can solve this issue by decoupling the pair of pre- and postconditions that are *local* and hold for the current continuation, from the *global* that hold for execution of whole functions in the environment and the result.

Heap Monotonicity. The reader familiar with Kripke logical relations might have already identified a different problem: the logical relation does not directly satisfy the Kripke monotonicity requirement—that is if two configurations are related then they should remain related for any future version of these configurations, as the evaluation of the expressions goes on. This allows to maintain relatedness of the environment of evaluation during execution of the configuration, and it is crucial to prove that closure conversion inhabits the logical relation. In our case, target heaps not only grow, but also shrink and become renamed, so our aim is to state that two related configurations remain related for all isomorphic environment-heap pairs:

$$\begin{gathered}\mathsf{FV}(e_1) \vdash (\sigma_1, H_1) \approx_{\beta_1} (\sigma_1', H_1') \Rightarrow \mathsf{FV}(e_2) \vdash (\sigma_2, H_2) \dot{\approx}_{\beta_2} (\sigma_2', H_2') \Rightarrow \\ \mathcal{C}^{(k,j)}\ \{P\}\ (H_1, \sigma_1, e_1)\ (H_2, \sigma_2, e_2)\ \{Q\} \Rightarrow \mathcal{C}^{(k,j)}\ \{P\}\ (H_1', \sigma_1', e_1)\ (H_2', \sigma_2', e_2)\ \{Q\}\end{gathered}$$

One could argue that it's enough to prove the semantics is deterministic up to heap isomorphism, in order to prove that our logical relation respects heap isomorphism. This is not the case: to do this we would have to impose additional requirements that the pre- and postcondition respect heap isomorphism. However, the concrete instantiations we are interested in are not preserved by heap isomorphism, because isomorphic heaps can have arbitrary sizes. Therefore, to achieve monotonicity, we explicitly close our logical definitions over all pairs of isomorphic environment-heap pairs.

7.2 Definition

With all this in mind, we can now simultaneously define a value relation that relates the results of evaluation and a configuration relation that relates the execution of two configurations. The relations are defined by induction on two indexes, the usual step index that indicates the maximum recursion depth, and the lookup index that indicates the maximum heap lookup depth.

Configuration Relation. The configuration relation in its final form appears below. Decoupling the indexes i and j is essential for the correctness proof—by quantifying over all lookup indexes we are able to relate the heaps at any depth for any given step index

$$\begin{aligned}
&\mathcal{C}^{(k,j)}\ \{P_G; P_L\}\ (H_1, \sigma_1, e_1)\ (H_2, \sigma_2, e_2)\ \{Q_G; Q_L\} \overset{\text{def}}{=} \\
&\forall H_1'\ \sigma_1'\ \beta_1\ H_2'\ \sigma_2'\ \beta_2\ r_1\ c_1\ m_1, \\
&\quad \mathsf{FV}(e_1) \vdash (\sigma_1, H_1) \approx_{\beta_1} (\sigma_1', H_1') \rightarrow \mathsf{FV}(e_2) \vdash (\sigma_2, H_2) \dot{\approx}_{\beta_2} (\sigma_2', H_2') \rightarrow \\
&\quad P_L\ (H_1', \sigma_1', e_1)\ (H_2', \sigma_2', e_2) \rightarrow c1 \leq k \rightarrow \\
&\quad H_1'; \sigma_1'; e_1 \Downarrow_{\mathtt{src}}^{(c_1, m_1)} r_1 \rightarrow \mathtt{not\_stuck}(H_1'; \sigma_1'; e_1) \rightarrow \\
&\quad \exists\ r_2\ c_2\ m_2\ \beta, \\
&\qquad H_2'; \sigma_2'; e_2 \Downarrow_{\mathtt{trg}}^{(c_2, m_2)} r_2 \wedge Q_L\ (H_1, \sigma_1, e_1)\ (c_1, m_1)\ (c_2,\ m_2) \wedge \mathcal{V}_{\beta}^{(k-c_1,j)}\ \{P_G\}\ r_1\ r_2\ \{Q_G\}
\end{aligned}$$

We additionally require that the source program cannot get stuck, that is, it terminates with any given amount of fuel either with a result or an out-of-time exception. This is needed to exclude programs that may time out with small fuel values but get stuck later.

Value Relation. The value relation (Figure 8) relates the result of two executions at a given step index i, lookup index j and location renaming function β, which keeps track of the mapping between source and target locations. Tracking the renaming allows us to establish that the reachable locations in the two heaps are in bijective correspondence.

Results are related as follows: An out-of-time exception relates only to an out-of-time exception. A pair of a location and a heap is related to another such pair if a.) both locations point to a constructed value in the heap, the target location agrees with the mapping of the source location of renaming function, and the arguments of the constructors are pairwise related at a strictly smaller lookup index; or b.) the source location points to a closure value and the target location points to a closure record. The closure environments must be related by the closure environment relation (denoted with $\mathcal{CL}$ and shown below). In addition, for any source and target heaps isomorphic to the original ones at the root set that contains the closure environment, the function maps logically related values to logically related results, and any strictly smaller step index. The environment part of the configuration is the environment part of the closure for the source, and singleton environment mapping the environment parameter to the appropriate location for the target, both extended with the appropriate bindings for the formal parameters and the recursive function. We additionally require that the precondition holds upon function entry. This is crucial in order to instantiate the precondition premise of the configuration relation in the application case.

$$\mathcal{V}_{\beta}^{(k,j)}\ \{P\}\ \mathtt{OOT}\ \mathtt{OOT}\ \{Q\} \overset{\text{def}}{=} \mathtt{True}$$

$$\mathcal{V}_{\beta}^{(k,j)}\ \{P\}\ (l_1, H_1)\ (l_2, H_2)\ \{Q\} \overset{\text{def}}{=} \qquad \text{if } H_1(l_1) = \mathsf{C}(\vec{v}_1) \text{ and } H_2(l_2) = \mathsf{C}(\vec{v}_2).$$
$$\beta(l_1) = l_2\ \wedge\ \forall\,(j' < j),\ \mathcal{V}_{\beta}^{(k,j')}\ \{P\}\ (\vec{v}_1, H_1)\ (\vec{v}_2, H_2)\ \{Q\}$$

$$\mathcal{V}_{\beta}^{(k,j)}\ \{P\}\ (l_1, H_1)\ (l_2, H_2)\ \{Q\} \overset{\text{def}}{=} \qquad \text{If } H_1(l_1) = \mathtt{Clo}(\mathtt{let\ rec}\ f_1\ \vec{x}\ =\ e_1, l_{\text{env}_1})$$
$$\text{and } H_2(l_2) = \mathsf{C}_{\text{CC}}(\mathtt{let\ rec}\ f_2\ \gamma :: \vec{x}\ =\ e_2, l_{\text{env}_2}).$$
$$\begin{array}{l}
\forall\,(j' < j),\ \mathcal{CL}_{\beta}^{(k,j')}\ \{P\}\ (l_{\text{env}_1}, H_1)\ (l_{\text{env}_2}, H_2)\ \{Q\}\ \wedge \\
\forall (i < k)\ H_1'\ \beta_1\ l_{\text{env}_1}'\ H_2'\ \beta_2\ l_{\text{env}_2}'\ \vec{v}_1\ \vec{v}_2, \\
\quad (l_{\text{env}_1}, H_1) \approx_{\beta_1} (l_{\text{env}_1}', H_1')\ \rightarrow\ (l_{\text{env}_2}, H_2) \approx_{\beta_2} (l_{\text{env}_2}', H_2')\ \rightarrow \\
\quad \mathtt{alloc}(H_1', \mathtt{Clo}(\mathtt{let\ rec}\ f_1\ \vec{x}\ =\ e_1, l_{\text{env}_1}')) = (l_{f_1}, H_1'')\ \rightarrow \\
\quad H_1'(l_{\text{env}_1}') = \mathsf{Env}(\sigma_{f_1})\ \rightarrow \\
\quad (\forall j,\ \mathcal{V}_{\beta_2 \circ \beta \circ \beta_1^{-1}}^{(k,j)}\ \{P\}\ (\vec{v}_1, H_1')\ (\vec{v}_2, H_2')\ \{Q\})\ \rightarrow \\
\quad P\ (H_1'', \sigma_1, e_1)\ (H_2', \sigma_2, e_2)\ \wedge \\
\quad (\forall j,\ \mathcal{C}^{(i,j)}\ \{P; P\ \}\ (H_1'', \sigma_1, e_1)\ (H_2', \sigma_2, e_2)\ \{Q; Q\})
\end{array}$$

where $\sigma_1 = \sigma_{f_1}[\vec{x} \mapsto \vec{v}_1][f_1 \mapsto l_{f_1}]$ and $\sigma_2 = [\gamma \mapsto l_{\text{env}_2}', \vec{x} \mapsto \vec{v}_2, f_2 \mapsto \mathtt{let\ rec}\ f_2\ \gamma :: \vec{x}\ =\ e_2]$.

$$\mathcal{V}_{\beta}^{(k,j)}\ \{P\}\ (l_1, H_1)\ (l_2, H_2)\ \{Q\} \overset{\text{def}}{=} \mathtt{False} \qquad \text{Otherwise.}$$

Fig. 8. Value relation

Closure Environment Relation. The closure relation mentioned by the above definition is defined simultaneously with the value relation and it relates a source and a target closure environment pointers. It asserts that the locations of the two pointers agree with the location renaming and that the values of each environment are pairwise related.

$$\begin{array}{l}
\mathcal{CL}_{\beta}^{(k,j')}\ \{P\}\ (l_1, H_1)\ (l_2, H_2)\ \{Q\} \overset{\text{def}}{=} \\
\qquad \exists \sigma, x_1, \ldots, x_n, v_1, \ldots, v_n,\ l_2 = \beta(l_1)\ \wedge\ H_1(l_1) = \mathsf{Env}(\sigma)\ \wedge\ H_2(l_2) = \mathsf{C}(v_1, \ldots, v_n)\ \wedge \\
\qquad\qquad \mathsf{dom}(\sigma) = \{x_1, \ldots, x_n\}\ \wedge\ \bigwedge^{i \in [1,n]} \mathcal{V}_{\beta}^{(k,j)}\ \{P\}\ (\sigma(x_i), H_1)\ (v_i, H_2)\ \{Q\}
\end{array}$$

Environment Relation. As usual, we lift the value relation to environments. The environment relation is annotated with a set of free variables and states that every variable in the set that is bound in the first environment, is also bound in the second environment and the bindings are logically related.

$$\mathcal{G}_{\beta}^{(k,j)}\ S \vdash \{P\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q\} \overset{\text{def}}{=}$$
$$\forall (x \in S)\ v_1,\ \sigma_1(x) = v_1 \to \exists\ v_2,\ \sigma_2(x) = v_2\ \wedge\ \mathcal{V}_{\beta}^{(k,j)}\ \{P\}\ (v_1, H_1)\ (v_2, H_2)\ \{Q\}$$

7.3 Properties

We discuss formally some important properties of the logical relation.

Relating Nonterminating Programs. A known limitation of logical relations is the inability to reason about divergence preservation. In the usual formulation of logical relations, two programs are vacuously related if the source program doesn't terminate (or gets stuck). We address this limitation here by adopting a fuel-based semantics and raising out-of-time exceptions, which as a result of the computation is logically related only with an other out-of-time exception. We show how this allows us to prove, for a certain class of postconditions, that the translation of a nonterminating program is also a nonterminating program, whose memory consumption is within the desired bounds.

We say a resource is *downward* (resp. *upward*) *f-bounded* if for the resource consumption of the source r_{src} and the target r_{trg} we have that $f(r_{\text{src}}) \leq r_{\text{trg}}$ (resp. $r_{\text{trg}} \leq f(r_{\text{src}})$) for some function f.

Theorem 7.2. *Assume $\mathcal{C}^{(k,j)}\ \{P_G; P_L\}\ (H_1, \sigma_1, e_1)\ (H_2, \sigma_2, e_2)\ \{Q_G; Q_L\}$ and that Q_L implies that time cost is downward f-bounded for some invertible function f, and that the memory consumption is upward g-bounded for some monotonic function g. Then if, for some m_1, $H_1; \sigma_1; e_1 \Uparrow_{\mathtt{src}}^{m_1}$ then $H_2; \sigma_2; e_2 \Uparrow_{\mathtt{trg}}^{m_2}$ for some m_2. Furthermore, if $m_1 \neq \infty$ then $m_2 \leq g(m_1)$.*

This theorem requires that the execution time of the source is bounded by the execution time of the target and the memory usage of the target is bounded by the memory usage of the source. The proof below gives some intuition about these requirements.

Proof. Consider any fuel value i. We will show that the target program diverges when run with fuel i, with some memory consumption that satisfies the bound. Since the source diverges, running the source program with $f^{-1}(i)$ must return an `OOT` exception with some memory consumption $m_1' \leq m_1$. From the logical relation, there exists a fuel j for which the target program also returns an `OOT` exception and has some memory consumption m_2'. Furthermore, from the postcondition, $f(f^{-1}(i)) \leq j$, that is $i \leq j$. Hence, by monotonicity of the resource consumption of the target evaluation semantics, the target program runs out of time when run with fuel i with some memory consumption $m_2'' \leq m_2'$.

To prove the memory bound, we derive from the post condition that $m_2' \leq g(m_1')$ and therefore $m_2'' \leq g(m_1')$. By monotonicity of g and the fact that $m_1' \leq m_1$ we obtain that $m_2'' \leq g(m_1)$. □

Local Reasoning. In its first simple form the logical relation did not support compositional reasoning, We address that by allowing a pair of local and a pair of global pre- and postconditions that allow us to restate compositional versions of the compatibility lemmas. Conceptually, by decoupling the global from the local conditions, we obtain monotonicity for the local conditions as that can vary without affecting the global invariant that holds for the execution of whole functions. As an example, we look again at the projection case for an example of a compatibility lemma.

First, we state strengthening and weakening properties for the pre- and postconditions. For compactness, we group constructors that are described as evaluation contexts together, although these properties are meaningful only when these contexts are singleton contexts.

Definition 7.3 (Precondition strengthening, context application). `PreCtx` $\mathcal{E}_1\ \mathcal{E}_2\ e_1\ e_2\ P_1\ P_2$ asserts that for any evaluation context interpretations $H_1; \sigma_1; \mathcal{E}_1 \blacktriangleright^{c_1} H_1'; \sigma_1'$ and $H_2; \sigma_2; \mathcal{E}_2 \blacktriangleright^{c_2} H_2'; \sigma_2'$, if

P_1 $(H_1, \sigma_1, \mathcal{E}_1[e_1])$ $(H_2, \sigma_2, \mathcal{E}_2[e_2])$ holds one the configuration before interpreting the context then P_2 (H_1', σ_1', e_1) (H_2', σ_2', e_2) holds one the final configurations.

Definition 7.4 (Postcondition weakening, context application). $\texttt{PostCtx}\ \mathcal{E}_1\ \mathcal{E}_2\ e_1\ e_2\ Q_1\ Q_2$ asserts that for any any evaluation context interpretations $H_1; \sigma_1; \mathcal{E}_1 \blacktriangleright^{c_1} H_1'; \sigma_1'$ and $H_2; \sigma_2; \mathcal{E}_2 \blacktriangleright^{c_2} H_2'; \sigma_2'$, if Q_2 (H_1', σ_1', e_1) (c, m) (c', m') on the final configuration and its resource consumption, then Q_1 $(H_1, \sigma_1, \mathcal{E}_1[e_1])$ $(c + c_1, \max(m, \texttt{size}_{\mathcal{R}}(H_1)[\texttt{FL}_{\texttt{Env}}(\sigma)[\texttt{FV}(\mathcal{E}_1[e_1])]]))$ $(c' + c_2, m')$ holds on the initial configuration and its resource consumption.

Above, the time consumption of the initial configurations is the time needed for evaluation of the final configurations, plus the time needed for the interpretation of the context. The memory consumption of the source is the maximum of the size of the reachable memory from the initial configurations and the memory consumption of the final configurations. The memory consumption of the target configuration is just propagated.

To establish the bounds when both programs timeout, we need to be able to derive the post condition from the precondition on the current configurations.

Definition 7.5 (Precondition entails postcondition, timeout).
$\texttt{PrePostTimeout}\ P\ Q$ asserts that if P (H_1, σ_1, e_1) (H_2, σ_2, e_2) and $c < \texttt{cost}(e_1)$ then
Q (H_1, σ_1, e_1) $(c, \texttt{size}_{\mathcal{R}}(H_1)[\texttt{FL}_{\texttt{Env}}(\sigma)[\texttt{FV}(e_1)]])$ $(c, \texttt{size}(H_2))$

We can now state and prove the projection compatibility theorem in its full generality, encompassing the weakening and strengthening rules for pre- and postconditions.

Theorem 7.6 (Projection Compatibility). *Assume that*

- $\texttt{PreCtx}\ (\texttt{let}\ x\ =\ y.j\ \texttt{in}\ [\,\cdot\,])\ (\texttt{let}\ x\ =\ y.j\ \texttt{in}\ [\,\cdot\,])\ e_1\ e_2\ P_L\ P_L'$
- $\texttt{PostCtx}\ (\texttt{let}\ x\ =\ y.j\ \texttt{in}\ [\,\cdot\,])\ (\texttt{let}\ x\ =\ y.j\ \texttt{in}\ [\,\cdot\,])\ e_1\ e_2\ Q_L\ Q_L'$
- $\texttt{PrePostTimeout}\ P_L\ Q_L$
- $\forall\, j,\ \mathcal{G}_\beta^{(k,j)}\ \{y\} \vdash \{P_G\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q_G\}$
- $\forall\, v_1\ v_2, \mathcal{V}_\beta^{(k,j)}\ \{P_G\}\ (v_1, H_1)\ (v_2, H_2)\ \{Q_G\} \rightarrow$
  $\mathcal{C}^{(k,j)}\ \{P_G; P_L'\}\ (H_1, \sigma_1[x \mapsto v_2], e_1)\ (H_2, \sigma_2[x \mapsto v_2], e_2)\ \{Q_G; Q_L'\}$

Then $\mathcal{C}^{(k,j)}\ \{P_G; P_L\}\ (H_1, \sigma_1, \mathcal{E}_1[\texttt{let}\ x\ =\ y.j\ \texttt{in}\ e_1])\ (H_2, \sigma_2, \mathcal{E}_2[\texttt{let}\ x\ =\ y.j\ \texttt{in}\ e_2])\ \{Q_G; Q_L\}$

Heap Size. The environment relation has some useful implications for the structure of the two heaps. First, it implies that the reachable portions of the two heaps (up to depth equal to the heap lookup index) are well-formed, meaning that there are no dangling pointers. Therefore, when we quantify over all lookup indexes in our proof we avoid having to carry around well-formedness assumptions, that are required for various lemmas and also the closure conversion proof. Second, and most importantly, it implies that the set of reachable locations of the two heaps are in correspondence up to the given renaming.

In our proof we will use this fact to show that directly after function entry (and garbage collection in the target) the sizes of the heaps are related. By quantifying the environment relation over all heap depths we can derive that the data structures in the reachable portions of two heaps are in 1-1 correspondence, up to the given renaming. Therefore, we can prove that if two environments are related $\forall\, j,\ \mathcal{G}_\beta^{(k,j)}\ S \vdash \{P\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q\}$, then the size of the reachable portion of the target is upper bounded by the size of the reachable portion of the source: $\texttt{size}_{\mathcal{R}}(H_2)[\texttt{FL}_{\texttt{Env}}(\sigma_2)[S]] <= \texttt{size}_{\mathcal{R}}(H_1)[\texttt{FL}_{\texttt{Env}}(\sigma_1)[S]]$. Notice the use of less than or equal here, since nothing prevents the renaming to map two source locations to the same (related with both) target location.

8 CORRECTNESS PROOF

With this machinery we can prove that closure conversion is correct. We prove that the source and target programs of closure conversion are logically related for an appropriate choice of pre- and postconditions. In this section we specify the concrete pre- and postconditions for the logical relations proof and we state the correctness theorem.

8.1 Time Bound

We are looking to find bounds for the execution time of the target. To handle divergent programs, the execution time of the closure converted program has to be bounded from below by a function of the execution time of the source. This is easy: closure-conversion only adds cost anyway. To derive an upper bound, consider the running time of the closure-converted program: for each source construct there will be an overhead associated with the handling of its free variables, which is at most three steps for each free variable. Function definitions incur additional overhead for the construction of the environment, which costs just one extra step (recall that source cost semantics accounts for cost equal to the number of the free variables of the functions, but this will be zero in the target since functions are closed). For applications there is also an additional overhead equal to three steps, associated with the projections of the code and environment components of the closures and the application of the extra parameter. For any source constructor, the overhead of the target is at most six steps. Therefore the cost of evaluating the target will be at most seven times the cost of evaluating the source. The relation we wish to establish for the execution cost of the two programs is $c_{\mathtt{src}} \leq c_{\mathtt{trg}} \leq k_{\mathtt{time}} * c_{\mathtt{trg}}$, where $k_{\mathtt{time}} = 7$.

8.2 Space Bound

To derive the space bound, consider how the sizes of the initial configurations relate immediately after function entry, and garbage collection in the target. The size of the target heap will be equal to its reachable portion, which will contain at most the space reachable from its original arguments and the space reachable from the environment argument (if the functions has free variables and the environment is used). The latter includes the allocated space for the environment itself, and the space reachable by the free variables of the original program, whose values are exactly the fields of the environment record. The reachable portion of the source heap will contain at least the space reachable by the arguments, and the space reachable by the free variables of the function, and the closure itself (code pointer + environment pointer + tag, 3 words) if the function is recursive. This closure will be allocated for the target closure too, but unlike the source semantics that does it eagerly, the target will do it just before the function name is used; so we must account for the cost of the closure that may be allocated later in the execution of the target. We also account for the size of the environment that may be present in the target Therefore, when a function starts executing the two initial configurations will be related as follows:

$$\mathtt{size}(H_{\mathtt{trg}})+3*|\{f\}\cap\mathsf{FV}(e_1)| \leq \mathtt{size}_{\mathcal{R}}(H_{\mathtt{src}})[\mathsf{FL}_{\mathsf{Env}}(\sigma_{\mathtt{src}})[\mathsf{FV}(e_{\mathtt{src}})]]+(1+|\mathsf{FV}(\mathtt{let}\,\mathtt{rec}\,f\,\vec{x} = e_{\mathtt{src}})|)$$

When evaluating the target function, the heap can grow at most by a an amount proportional to the size of $e_{\mathtt{src}}$, until the program returns or the next function is called and this relation in established again. Let $\mathtt{cost}^{\mathtt{space}}_{\mathtt{exp}}(e_{\mathtt{src}})$ be the maximum amount of allocation that can happen during the evaluation of $e_{\mathtt{trg}}$. The size of the target heap, during the evaluation of the closure converted function, will remain below $\mathtt{size}_{\mathcal{R}}(H_{\mathtt{src}})[\mathsf{FL}_{\mathsf{Env}}(\sigma_{\mathtt{src}})[\mathsf{FV}(e_{\mathtt{src}})]] + (1 + |\mathsf{FV}(\mathtt{let}\,\mathtt{rec}\,f\,\vec{x} = e_{\mathtt{src}})|) + \mathtt{cost}^{\mathtt{space}}_{\mathtt{exp}}(e_{\mathtt{src}})$. Hence the memory consumption of the program, will be upper bounded by the maximum of the above expression and the memory consumption of the execution of its continuation, which is a function call and has a similar upper bound. Observe that the size of the reachable portion of the source heap will be below the memory consumption $m_{\mathtt{src}}$, which follows directly by the way the source space consumption is calculated. Therefore we expect that the target space consumption will be below the source consumption plus the maximum overhead incurred by the evaluation of the current function or any future function call.

$$m_{\mathsf{trg}} \leq m_{\mathsf{src}} + \max(1 + |\mathsf{FV}(\mathtt{let\ rec}\ f\ \vec{x}\ =\ e_{\mathsf{src}})| + \mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e_{\mathsf{src}}),\quad \mathtt{cost}^{\mathsf{space}}_{\mathsf{heap}}(H_{\mathsf{trg}}))$$

The function $\mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(\cdot)$ captures the maximum amount of allocation that can happen either during the evaluation of the closure-conversion of its argument, or during the evaluation of the closure-conversion of a function nested ins nested inside it. It is defined as,

$$\begin{array}{rcl}
\mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(\mathtt{let}\ x\ =\ \mathsf{C}(\vec{y})\ \mathtt{in}\ e) & \stackrel{\text{def}}{=} & 1 + |\vec{y}| + \mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e) \\
\mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(\mathtt{let}\ x\ =\ y.j\ \mathtt{in}\ e) & \stackrel{\text{def}}{=} & \mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e) \\
\mathtt{case}\ y\ \mathtt{of}\ \{\mathsf{C}_i\ \longrightarrow\ e_i\}_{i\in I} & \stackrel{\text{def}}{=} & \max\limits_{i\in I}(\mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e_i)) \\
\mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(\mathtt{let\ rec}\ f\ \vec{x}\ =\ e_1\ \mathtt{in}\ e_2) & \stackrel{\text{def}}{=} & \max(|\mathsf{env}| + 3 + \mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e_2), |\mathsf{env}| + \mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e_1)) \\
 & & \text{where } |\mathsf{env}| = 1 + |\mathsf{FV}(\mathtt{let\ rec}\ f\ \vec{x}\ =\ e_1)| \\
\mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(f\ \vec{x}) & \stackrel{\text{def}}{=} & 0 \\
\mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(\mathtt{halt}(x)) & \stackrel{\text{def}}{=} & 0
\end{array}$$

Most cases are straightforward, but it is worth discussing the function definition case, which is the most complicated. We take the maximum of two terms that correspond to the maximum allocation that can happen during the evaluation of the current expression, and the maximum allocation that can happen during the evaluation of function being defined. The first term amounts to the size of the closure environment that will be allocated, the space of the closure that will be allocated in when it is used in this scope, and the space that will be allocated when evaluating the rest of the program. The second term amounts to the size of the environment of the function plus the space that will be allocated during its evaluation. We can compute the maximum amount of allocation that can happen when evaluating any function pointer that is stored in the heap as described by the following definition.

$$\mathtt{cost}^{\mathsf{space}}_{\mathsf{heap}}(H) \stackrel{\text{def}}{=} \max\{\mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e) + (1 + |\mathsf{FV}(\mathtt{let\ rec}\ f\ \vec{x}\ =\ e)|) \mid \exists l,\ H(l) = \mathtt{let\ rec}\ f\ \vec{x}\ =\ e\}$$

We have formulated the bound that we expect to hold for the space consumption of the two programs, but we're not done yet. These pre- and postconditions will hold only directly after function entry, not necessarily during the execution of the two functions. We need to find pre- and postconditions that capture the relation of the resources at any point during execution. Let us first state the precondition in a more general way. In particular, assume that for some parameters A and δ we have that $\mathtt{size}(H_{\mathsf{trg}}) + 3 * |F| \leq A + \delta$, where F are the function names in scope that have not been used yet, hence the target code has not constructed their closures and therefore we have to account for their future allocation. Conceptually, A stands for the size of the reachable source heap at the function entry point and δ is the extra space that has been allocated during the execution of the target code after the function entry point. If the initial configurations are related this way, then the postcondition that bounds the space consumption of the target is,

$$m_{\mathsf{trg}} \leq \max(A + \mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e_{\mathsf{src}}) + \delta, m_{\mathsf{src}} + \max(\mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e_{\mathsf{src}}),\quad \mathtt{cost}^{\mathsf{space}}_{\mathsf{heap}}(H_{\mathsf{src}})))$$

Intuitively, the term $A + \mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e_1) + \delta$ captures the maximum space consumption for the execution of the current function, and the term $m_{\mathsf{src}} + \max(\mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e_{\mathsf{src}}), \mathtt{cost}^{\mathsf{space}}_{\mathsf{heap}}(H_{\mathsf{src}}))$ the maximum space consumption of any future execution of a function that is either defined in e_{src} or is a heap pointer.

We can now formulate the concrete pre- and postconditions that we wish to establish when proving closure conversion correct. The local pre- and postconditions are,

$$\mathsf{P_L}\ F\ A\ \delta\ (H_{\mathsf{src}}, \sigma_{\mathsf{src}}, e_{\mathsf{src}})\ (H_{\mathsf{trg}}, \sigma_{\mathsf{trg}}, e_{\mathsf{trg}}) \overset{\text{def}}{=} \mathtt{size}(H_{\mathsf{trg}}) + 3 * |F| \le A + \delta$$

$$\begin{aligned}\mathsf{Q_L}\ A\ \delta\ (H_{\mathsf{src}}, \sigma_{\mathsf{src}}, e_{\mathsf{src}})\ (c_{\mathsf{src}}, m_{\mathsf{src}})\ (c_{\mathsf{trg}}, m_{\mathsf{trg}}) &\overset{\text{def}}{=} c_{\mathsf{src}} \le c_{\mathsf{trg}} \le k_{\mathtt{time}} * c_{\mathsf{trg}} \land \\ m_{\mathsf{trg}} \le \max(A + \mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e_{\mathsf{src}}) + \delta, m_{\mathsf{src}} &+ \max(\mathtt{cost}^{\mathsf{space}}_{\mathsf{exp}}(e_{\mathsf{src}}), \mathtt{cost}^{\mathsf{space}}_{\mathsf{heap}}(H_{\mathsf{src}})))\end{aligned}$$

Note that the local pre- and postconditions are parameterized by A, δ, and F which will also be parameters to the proof of closure conversion.

The global pre- and postconditions are instances of the local ones that hold when considering the execution of whole functions. Hence, the parameter F will be instantiated with a singleton set containing the name of the function, have only one function name (the current, δ with the size of the function's environment, and A with the size of the reachable portion of the heap upon function entry. The parameter F is the set of free variables that will be passed by the logical relation (not shown in its definition) to calculate the size of the closure environment.

$$\begin{aligned}&\mathsf{P_G}\ F\ (H_{\mathsf{src}}, \sigma_{\mathsf{src}}, e_{\mathsf{src}})\ (H_{\mathsf{trg}}, \sigma_{\mathsf{trg}}, e_{\mathsf{trg}}) \overset{\text{def}}{=} \\ &\quad \mathsf{P_L}\ F\ \mathtt{size}_{\mathcal{R}}(H_{\mathsf{src}})[\mathsf{FL_{Env}}(\sigma_{\mathsf{src}})[\mathsf{FV}(e_{\mathsf{src}})]]\ (1 + |F|)\ (H_{\mathsf{src}}, \sigma_{\mathsf{src}}, e_{\mathsf{src}})\ (H_{\mathsf{trg}}, \sigma_{\mathsf{trg}}, e_{\mathsf{trg}})\end{aligned}$$

$$\begin{aligned}&\mathsf{Q_G}\ F\ (H_{\mathsf{src}}, \sigma_{\mathsf{src}}, e_{\mathsf{src}})\ (c_{\mathsf{src}}, m_{\mathsf{src}})\ (c_{\mathsf{trg}}, m_{\mathsf{trg}}) \overset{\text{def}}{=} \\ &\quad \mathsf{Q_L}\ \mathtt{size}_{\mathcal{R}}(H_{\mathsf{src}})[\mathsf{FL_{Env}}(\sigma_{\mathsf{src}})[\mathsf{FV}(e_{\mathsf{src}})]]\ (1 + |F|)\ (H_{\mathsf{src}}, \sigma_{\mathsf{src}}, e_{\mathsf{src}})\ (c_{\mathsf{src}}, m_{\mathsf{src}})\ (c_{\mathsf{trg}}, m_{\mathsf{trg}})\end{aligned}$$

8.3 Correctness

The main correctness theorem states that the input and the output programs of closure conversion are related when evaluated in related heap-environment pairs. Our logical relation implies semantic preservation and establishes the resource bounds. In our proofs, we make the explicit assumption that the source program has unique bindings which are also disjoint from the free variables of the program, but leave these assumptions implicit in the paper presentation of the main theorem. As usual, in the fundamental theorem of the logical relation, we require that the environments are logically related, and we maintain this invariant throughout the proof. But in the case of closure conversion this relation holds only for the local environments, that contain parameters of the current function and locally defined variables, excluding newly defined functions that have not been used yet (because the closure-converter will not build the closure for a function until its first occurrence).

Recall the closure conversion judgment: $\Gamma, \Phi \vdash_{(\phi,\gamma)} e \rightsquigarrow \bar{e}$. The environment Γ corresponds to the local variables and hence its contents will be related by the environment relation. The rest of the evaluation environment consists of the function names that have been defined locally but have not been used yet (given by the set $\mathsf{dom}(\phi) \setminus \Gamma$)—hence the target code has not constructed their closures yet—and the free variables that are not local to the current scope (given by the global environment Φ) which correspond (in the target evaluation environment) to what is stored in the closure environment parameter.

Two additional environment invariants capture these relations for these parts of the environment. The first relation asserts that the variables in the global environment of the source program are logically related with the values stored in the closure environment of the target program.

$$\begin{aligned}&\mathcal{FV}^{(k,j)}_{\beta}\ \Phi; \gamma \vdash \{P\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q\} \overset{\text{def}}{=} \\ &\quad \exists x_1, \ldots, x_n, v_1, \ldots, v_n,\ \Phi = [x_1, \ldots, x_n] \ \land\ H_2(\sigma_2(\gamma)) = \mathsf{C}(v_1, \ldots, v_n) \land \\ &\qquad \textstyle\bigwedge^{i \in [1,n]} \mathcal{V}^{(k,j)}_{\beta}\ \{P\}\ (\sigma(x_i), H_1)\ (v_i, H_2)\ \{Q\}\end{aligned}$$

In the same spirit we formulate a relation that connects the closures in the source with the closures that have not yet been constructed in the target.

$$\begin{array}{l}\boldsymbol{\mathcal{F}}_{\beta}^{(k,j)}\ \Gamma;\phi \vdash \{P\}\ (\sigma_1, H_1)\ (\sigma_2, H_2)\ \{Q\} \stackrel{\text{def}}{=} \\ \quad \forall\ (f \in \mathsf{dom}(\phi) \setminus \Gamma)\ l_f\ H_2', \quad \mathtt{alloc}(H_2, \mathsf{C}_{\mathsf{CC}}(\sigma_2(f); \sigma_2(\phi(f)))) = (l_f, H_2') \quad \Rightarrow \\ \qquad \boldsymbol{\mathcal{V}}_{\beta}^{(k,j)}\ \{P\}\ (\sigma_1(f), H_1)\ (l_f, H_2')\ \{Q\}\end{array}$$

The above relation asserts that the value of source environment for any of the function names in $\mathsf{dom}(\phi) \setminus \Gamma$ is logically related to a location that points to a freshly constructed target closure, whose code component is the value of the function name in the target environment, and the environment component is the name ($\phi(f)$) of the closure environment that corresponds to f in the target environment. Maintaining this invariant allows us to reestablish the local environment relatedness after the target closure has been constructed and the function name has been added to the local environment.

Using the above invariants we can state and prove the fundamental theorem of our logical relation, that the closure conversion relation inhabits the logical relation for our choice of pre- and postconditions. We additionally require that the location renaming between the source and target heaps is injective at the reachable portion of its domain order to establish that the reachable portions of the two heaps are in bijection.

Theorem 8.1 (Correctness of Closure Conversion). *Assume that*

- $\Gamma, \Phi \vdash_{(\phi,\gamma)} e \rightsquigarrow \bar{e}$
- $\forall\, j,\quad \boldsymbol{\mathcal{G}}_{\beta}^{(k,j)}\ \Gamma \vdash \{\mathsf{P_G}\}\ (\sigma_{\mathsf{src}}, H_{\mathsf{src}})\ (\sigma_{\mathsf{trg}}, H_{\mathsf{trg}})\ \{\mathsf{Q_G}\}$
- $\forall\, j,\quad \boldsymbol{\mathcal{FV}}_{\beta}^{(k,j)}\ \Phi;\gamma \vdash \{\mathsf{P_G}\}\ (\sigma_{\mathsf{src}}, H_{\mathsf{src}})\ (\sigma_{\mathsf{trg}}, H_{\mathsf{trg}})\ \{\mathsf{Q_G}\}$
- $\forall\, j,\quad \boldsymbol{\mathcal{F}}_{\beta}^{(k,j)}\ \Gamma;\phi \vdash \{\mathsf{P_G}\}\ (\sigma_{\mathsf{src}}, H_{\mathsf{src}})\ (\sigma_{\mathsf{trg}}, H_{\mathsf{trg}})\ \{\mathsf{Q_G}\}$

and let $F = \mathsf{dom}(\phi) \setminus \Gamma$.
Then for all $j\ A\ \delta$, $\boldsymbol{\mathcal{C}}^{(k,j)}\ \{\mathsf{P_G}; \mathsf{P_L}\ F\ A\ \delta\}\ (H_{\mathsf{src}}, \sigma_{\mathsf{src}}, e)\ (H_{\mathsf{trg}}, \sigma_{\mathsf{trg}}, \bar{e})\ \{\mathsf{Q_G}; \mathsf{Q_L}\ A\ \delta\}$.

Doing the proof amounts to showing that the concrete bounds satisfy the compatibility requirements, then applying the compatibility properties of the logical relation to relate the contexts generated by the closure conversion with those of the source program, and finally, for the inductive cases of the proof, applying the inductive hypothesis (the proof is by induction on the step-index and then by case analysis on the expression). The application case is almost immediate using the environment relation. The abstraction case requires us to show that the closures we are allocating are related which we obtain from the induction hypothesis and the fact that the size bound holds upon function entry.

Now consider a complete program, one without free variables. If it terminates, the closure-converted program terminates, the result is correct, and the resource bound is satisfied: it is safe for space and safe for time.

Corollary 8.2 (Correctness of Closure Conversion, top-level, termination).
If $e \rightsquigarrow \bar{e}$ *and* $e \Downarrow_{\mathsf{src}}^{(c_1,m_1)} (v_1, H_1)$, *then there exist* v_2, H_2, c_2, m_2, β *such that a.) the target terminates* $\bar{e} \Downarrow_{\mathsf{trg}}^{(c_2,m_2)} (v_2, H_2)$, *b.) the results are related* $\forall\ i\ j$, $\boldsymbol{\mathcal{V}}_{\beta}^{(k,j)}\ \{\mathsf{P_G}\}\ (v_1, H_1)\ (v_2, H_2)\ \{\mathsf{Q_G}\}$, *and c). the resource consumption for time and space is preserved:* $c_1 \le c_2 \le k_{\mathtt{time}} * c_1$ *and* $m_2 \le m_1 + \mathtt{cost}_{\mathtt{exp}}^{\mathtt{space}}(e) + 1$.

If the source program diverges, the closure converted program must also diverge, and if the source program runs in bounded space then the target must run in correspondingly bounded space.

Corollary 8.3 (Correctness of Closure Conversion, top-level, divergence). *If $e \rightsquigarrow \bar{e}$ and $e \Uparrow^{m_1}_{\mathsf{src}}$ then there exists m_2 such that the target diverges $\bar{e} \Uparrow^{m_2}_{\mathsf{trg}}$ and $m_2 \leq m_1 + \mathsf{cost}^{\mathsf{space}}_{\mathsf{exp}}(e) + 1$.*

Coq Development. Our results have all been fully formalized in the Coq proof assistant. The length of the main closure conversion proof is 1855 lines and the logical relation framework along with the compatibility lemmas is 1815 lines of specification code and 1921 lines of proof code.

Compositionality. We have presented a resource-safety proof for closure conversion. For a provably resource-preserving compiler we have to establish this result for all the passes of the compiler and compose the proofs to get and end-to-end theorem. Unfortunately, composition cannot happen at the level of logical relations.

Consider the symmetric version of this logical relation (in which function calls are the same in the source and target programs). This relation is, in fact, an equivalence relation, and one can verify multiple passes individually and derive that the composition of the passes inhabits the logical relation. However, the "symmetric version" does not compose with the closure-conversion logical relation and therefore the reasoning can not be carried out end-to-end at the level of logical relations. Composition can still happen at the level of top-level corollaries, using a weaker notion of value refinement, and applies only to whole-program compilation.

9 RELATED WORK

Space-safety of Program Transformations. Others have already observed that program transformations ought to be safe with respect to their resource consumption. Minamide [1999] proves space-safety of the CPS transformation by showing, using a forward simulation argument, that CPS preserves the size of the reachable portion in the heap within a constant factor.

Improvement theory [Sands 1992, 1998] can be used to characterize transformations guaranteed to never worsen the asymptotic complexity of a program with respect to a particular resource. A program is called an improvement of another if its execution is more efficient with respect to a particular resource in any given program context. To show that a transformation is resource-safe it suffices to show that local steps of the transformation inhabit a particular improvement relation. This technique has been studied both in the context of time-safety and space-safety [Gustavsson and Sands 1999, 2001]. Theory of space improvement has only been applied to local transformations.

Resource Bound Certification. Source resource consumption bounds can be certified by showing that they are preserved through the compilation pipeline. Crary and Weirich [2000] present a decidable type system capable of establishing upper bounds on resource consumption of programs and they show that these bounds are preserved all the way to assembly language. They do so by introducing a virtual clock that winds down at every function application. Although this technique suffices to show preservation of time, it does not scale to space usage in a garbage collected setting. Since memory does not grow monotonically, "ticks" cannot capture space consumption.

Certified space-preserving compilers for imperative languages exist as well, but employ proof techniques that do not directly apply to functional programs and higher-order state. Carbonneaux et al. [2014] prove that stack-space bounds for C programs are preserved through CompCert by extending the trace preservation proof of the compiler to also prove stack space consumption. Amadio and Régis-Gianas [2012] show that the *labelling method*, an instrumentation technique for monitoring resources the program consumes, is preserved through a compilation chain. The labels of the source program can be used to reason about its execution time, allowing for end-to-end bound certification. Besson et al. [2017] extend the semantics and the proof of CompCert to verify that memory consumption is preserved.

Source Cost Analysis. Source-level cost analysis is complementary to the work presented in this paper. Most related to our work are techniques that apply to higher-order functional languages and support time and space usage [Jost et al. 2010], and garbage collection [Albert et al. 2010; Unnikrishnan and Stoller 2009].

Relational cost analysis is technically related to our work, as it relates the resource consumption of the execution of two programs. Çiçek *et al.* 2017; 2016; 2015 develop type-and-effect refinement type systems capable of establishing upper bounds for the resource consumption of two programs (or two executions of the same program), and their applications include analysis of the cost of incremental computation and reasoning about side-channel freedom. The technical background of this line of work is similar to ours and we drew inspiration from it when designing this framework. The soundness proof of the type systems uses a logical relation that relates the final values of the computation, as well as the resource consumption of the two programs.

Garbage Collection Specification. Our formalization of garbage collection is inspired by the one presented by Morrisett and Harper [1997] in their formalization of an abstract machine for a functional language that allows the heap to be garbage collected nondeterministically. Our specification of garbage collection is similar to theirs, with the addition that we require the heap to be fully garbage collected which is require to prove the bounds.

Logical Relations. The applications of logical relations to correctness of program transformations have been studied widely. Our logical relation is unusual in that is supports reasoning in presence of garbage collection. Hur and Dreyer [2011] observed that Kripke logical relations are not directly compatible with garbage collection because of the heap monotonicity requirement that is violated in presence of garbage collection. They address that by the means of *logical memories* in which locations are never deallocated. Logical memories are connected to physical memories by a lookup table that specifies whether a location is live in the physical memory and which physical location it corresponds. Our solution is to close our relation over all isomorphic heaps, hence guaranteeing that related computations will be related for future, possibly garbage-collected, heaps.

Our notion of global vs. local conditions resembles the concepts of *local* and *global knowledge* in parametric bisimulations [Hur et al. 2012; Neis et al. 2015]. Local knowledge, similar to local pre- and postconditions in our case, represents the relations known to hold locally during the proofs. In contrast, global knowledge characterizes what values are equivalent in general, much as our global conditions characterize what is globally true about the execution of programs.

10 CONCLUSION

We have presented the first formal proof that closure conversion with flat closure representation is safe for space (as well as correct with respect to evaluation semantics). To do so we devised a logical relation that allows us to reason compositionally about intensional properties of programs along with extensional ones. Compared to known limitations of standard logical relations, our new relation gives the ability to reason about resource preservation in presence of garbage collected heaps, and the reasoning can be extended to diverging source programs.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grants CCF-1407794 and CCF-1521602. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. 2010. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM '10)*. ACM, New York, NY, USA, 121–130. https://doi.org/10.1145/1806651.1806671

Roberto M. Amadio and Yann Régis-Gianas. 2012. Certifying and Reasoning on Cost Annotations of Functional Programs. In *Proceedings of the Second International Conference on Foundational and Practical Aspects of Resource Analysis (FOPARA'11)*. Springer-Verlag, Berlin, Heidelberg, 72–89. https://doi.org/10.1007/978-3-642-32495-6_5

Abhishek Anand, Andrew W. Appel, John Gregory Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq : A verified compiler for Coq. In *CoqPL'17: The Third International Workshop on Coq for Programming Languages*. 2.

Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press, New York.

Andrew W. Appel and Marcelo J. R. Gonçalves. 1993. *Hash-consing garbage collection*. Technical Report CS-TR-412-93. Princeton University Department of Computer Science.

Andrew W. Appel and Trevor Jim. 1989. Continuation-Passing, Closure-Passing Style. In *16th ACM Symp. on Principles of Programming Languages*. ACM Press, New York, 293–302.

Andrew W. Appel and Zhong Shao. 1996. Empirical and Analytic Study of Stack Versus Heap Cost for Languages with Closures. *J. Funct. Program.* 6 (01 1996), 47–74. https://doi.org/10.1017/S095679680000157X

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2017. CompCertS: A Memory-Aware Verified C Compiler using Pointer as Integer Semantics. In *ITP 2017 - 8th International Conference on Interactive Theorem Proving (ITP 2017: Interactive Theorem Proving)*, Vol. 10499. Springer, Brasilia, Brazil, 81–97. https://doi.org/10.1007/978-3-319-66107-0_6

Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end Verification of Stack-space Bounds for C Programs. *SIGPLAN Not.* 49, 6 (June 2014), 270–281. https://doi.org/10.1145/2666356.2594301

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 316–329. https://doi.org/10.1145/3009837.3009858

Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. 2016. A Type Theory for Incremental Computational Complexity with Control Flow Changes. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 132–145. https://doi.org/10.1145/2951913.2951950

Ezgi Çiçek, Deepak Garg, and Umut Acar. 2015. Refinement Types for Incremental Computational Complexity. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer, Heidelberg, 406–431.

Karl Crary and Stephanie Weirich. 2000. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*. ACM, New York, NY, USA, 184–198. https://doi.org/10.1145/325694.325716

Amer Diwan, Eliot Moss, and Richard Hudson. 1992. Compiler Support for Garbage Collection in a Statically Typed Language. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/143095.143140

Vyacheslav Egorov. 2012. Grokking V8 closures for fun (and profit?). https://mrale.ph/blog/2012/09/23/grokking-v8-closures-for-fun.html/.

David Glasser. 2013. An interesting kind of JavaScript memory leak. https://blog.meteor.com/an-interesting-kind-of-javascript-memory-leak-8b47d2e7f156/.

Jörgen Gustavsson and David Sands. 1999. A Foundation for Space-Safe Transformations of Call-by-Need Programs. *Electronic Notes in Theoretical Computer Science* 26 (1999), 69 – 86. https://doi.org/10.1016/S1571-0661(05)80284-1 HOOTS '99, Higher Order Operational Techniques in Semantics.

Jörgen Gustavsson and David Sands. 2001. Possibilities and Limitations of Call-by-need Space Improvement. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. ACM, New York, NY, USA, 265–276. https://doi.org/10.1145/507635.507667

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 359–373. https://doi.org/10.1145/3009837.3009842

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke Logical Relation Between ML and Assembly. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1926385.1926402

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The Marriage of Bisimulations and Kripke Logical Relations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 59–72. https://doi.org/10.1145/2103656.2103666

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-order Programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages (POPL '10). ACM, New York, NY, USA, 223–236. https://doi.org/10.1145/1706299.1706327

Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. 2012. Optimizing Closures in O(0) Time. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming (Scheme '12)*. ACM, New York, NY, USA, 30–35. https://doi.org/10.1145/2661103.2661106

Richard Kelsey and Paul Hudak. 1989. Realistic Compilation by Program Transformation. In *16th ACM Symp. on Principles of Programming Languages*. ACM Press, New York, 281–92.

Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *POPL'14: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 179–191.

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.

Andrew McCreight, Tim Chevalier, and Andrew Tolmach. 2010. A Certified Framework for Compiling and Executing Garbage-collected Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 273–284. https://doi.org/10.1145/1863543.1863584

Yasuhiko Minamide. 1999. Space-Profiling Semantics of the Call-by-Value Lambda Calculus and the CPS Transformation. *Electr. Notes Theor. Comput. Sci.* 26 (1999), 105–120. https://doi.org/10.1016/S1571-0661(05)80286-5

Greg Morrisett and Robert Harper. 1997. Semantics of Memory Management for Polymorphic Languages. In *Higher Order Operational Techniques in Semantics, Publications of the Newton Institute*. Cambridge University Press, 175–226.

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A Compositionally Verified Compiler for a Higher-order Imperative Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 166–178. https://doi.org/10.1145/2784731.2784764

David Sands. 1992. Operational Theories of Improvement in Functional Languages. In *Functional Programming, Glasgow 1991*, Rogardt Heldal, Carsten Kehler Holst, and Philip Wadler (Eds.). Springer London, London, 298–311.

David Sands. 1998. Improvement Theory and Its Applications. In *Higher Order Operational Techniques in Semantics*, Andrew D. Gordon and Andrew M. Pitts (Eds.). Cambridge University Press, New York, NY, USA, 275–306. http://dl.acm.org/citation.cfm?id=309656.309672

Olivier Savary Bélanger and Andrew W. Appel. 2017. Shrink Fast Correctly!. In *PPDP'17: International Symposium on Principles and Practice of Declarative Programming*. ACM Press, 49–60.

Zhong Shao and Andrew W. Appel. 1994. Space-efficient Closure Representations. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming (LFP '94)*. ACM, New York, NY, USA, 150–161. https://doi.org/10.1145/182409.156783

Zhong Shao and Andrew W. Appel. 2000. Efficient and Safe-for-space Closure Conversion. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 129–161. https://doi.org/10.1145/345099.345125

Leena Unnikrishnan and Scott D. Stoller. 2009. Parametric Heap Usage Analysis for Functional Programs. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09)*. ACM, New York, NY, USA, 139–148. https://doi.org/10.1145/1542431.1542451

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 79 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133903