12-2019

Formally Verified Space-Safety for Program Transformations

Jason Carr
jac3559@rit.edu

Formally Verified Space-Safety for Program Transformations

by

Jason Carr

THESIS

Presented to the Faculty of the Department of Computer Science

Golisano College of Computer and Information Sciences

Rochester Institute of Technology

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

Rochester Institute of Technology

December 2019

Formally Verified Space-Safety for Program Transformations

APPROVED BY

SUPERVISING COMMITTEE:

Matthew Fluet, Supervisor

Ivona Bezáková, Reader

Stanisław Radziszowski, Observer

Abstract

Formally Verified Space-Safety for Program Transformations

Jason Carr, M.S.
Rochester Institute of Technology, 2019

Supervisor: Matthew Fluet

Existing work on compilers has often primarily concerned itself with preserving behavior, but programs have other facets besides their observable behavior. We expect that the performance of our code is preserved and bettered by the compiler, not made worse. Unfortunately, that's exactly what sometimes occurs in modern optimizing compilers. Poor representations or incorrect optimizations may preserve the correct behavior, but push that program into a different complexity class entirely. We've seen such blowups like this occurring in practice, and many transformations have pitfalls which can cause issues. Even when a program is not dramatically worsened, it can cause the program to use more resources than expected, causing issues in resource-constrained environments, and increasing garbage-collection pauses. While several researchers have noticed potential issues, there have been a relative dearth of proofs for space-safety, and none at all concerning non-local optimizations.

This work expands upon existing notions of space-safety, allowing them to be used to reason about long-running programs with both input and output, while ensuring that the program maintains some temporal locality of space costs. In addition, this work includes new proof techniques which can handle more dramatic shifts in the program and heap structure than existing methods, as well as more frequent garbage collection. The results are formalized in Coq, including a proof of space-safety for lifting data up in scope, which increases sharing and saves duplicate work, but may also catastrophically increase space usage, if done incorrectly.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Problem

While it is common to prove correctness of program transformations, in terms of results, efficiency of time and space may often be neglected. Verified compilation has grown in importance in recent years; several formally verified compilers, such as CompCert and CakeML have arisen. Still, mere correctness of extrinsic semantics is not always enough. A program that exceeds all available memory has no opportunity to be correct, so a proof of correctness of behavior, with no regards to space safety, is not always sufficient. Formally proving efficiency may be even more important for a general-purpose compiler: It is easy for users to notice failures of extensional correctness, but understanding a performance bug requires being intimately aware of the performance of a piece of code, and the possibility of compiler performance bugs. This is quite a difficult task for high-level code, especially for languages with little performance specification. This work will formalize space-safety proofs, a large step towards guaranteed performance and correctness for compilers.

Various researchers have noticed this problem. Chase noticed that several allocation optimizations that allowed objects to appear on the stack could asymptotically increase space usage [2]. Appel and Shao showed that some closure representations could cause space leaks [1,24]. A vast number of representation choices and optimizations may cause higher space usage than is necessary. Several apparent optimizations, such hoisting definitions out of lambdas, performing common subexpression elimination may cause issues. Failure to collect objects referenced from the stack in a timely manner may also cause problems. These effects appear in practice: When the Standard ML of New Jersey (SML/NJ) compiler was changed to use Appel's suggested, more space-efficient, closure representation, a few programs used as much as 80× less space with many more having less significant gains [24]. The Glasgow Haskell Compiler's lambda lifting pass “Full laziness” does not distinguish between space-safe and space-unsafe lifts. This can cause major space leaks in practice [13]. A flag was added to allow disabling it, but this is all-or-nothing. If we are to believe in the difficulty of noticing performance regressions, many more effects may have been discounted: Users may attempt to change their code, or simply allocate more resources. In the MLton optimizing compiler, which has inspired much of this development, space-unsafe optimizations have previously caused problems; indeed one discovery was made by Spoonhower et al.[25]. They developed a formal cost model and compared it to real programs, but noticed that it was not upheld by the compiler; specifically, an unsafe flattening operation had kept large datastructures live for too long, resulting in quadratic space complexity, instead of linear.

Although it is common to consider time as preferable to space, the impacts of space usage may be significant but hard to measure. In general, more uncollectable space usage means more frequent, and slower garbage collection pauses. Furthermore, larger data hinders efficient usage of CPU caches, potentially making for massive slowdowns with small changes. Although memory is often available for servers and desktops, the range of computing devices is growing. High-level languages need not be shunned in environments such as “Internet of Things” devices, webpages, embedded environments, if we are careful to ensure their performance. A clear performance semantics is necessary in order to reason accurately about resulting code.

1.2 Contributions of this Work

This work expands and strengthens the notion of space-safety to give stronger guarantees and apply to more programs. It will furthermore include a novel proof of efficiency using this strengthened definition.

This work has several unique contributions:

- Previous definitions considered only the maximum space usage for a full execution. This work proves space-safety locally for an abstract machine with input and output. Thus, one large allocation does not give a future grant for wastefulness.
- This language includes non-deterministic input and output, so it is sufficient for safety of long-running, interactive programs. In particular, it proves space safety for programs which can use unbounded space, which has not been proven before.
- The proof for lifting/globalization includes both a permutation of allocation order, and a sharing of previously unshared objects, requiring different proof techniques.
- The work has been formally verified in Coq [11].

Chapter 2

Background

2.1 Space semantics

For a Turing machine, measuring space usage is quite simple: one can simply count the maximum number of tape squares needed. But, for a complex programming language with procedures, objects, and garbage collection, the case is not so clear. A look at models for the lambda calculus may help enlighten this.

Standard substitution rules for the lambda calculus would indicate simple substitution of values into terms. The space can then be measured by the total size of the term. This model is rather unrealistic to the expectations of programmers: if a large object is bound to a variable, which is then used in three places, this model would have copied it to each of the three use sites. Rather, we should hope that the object is shared. When an object is created, we can instead assign it an address, which will be placed in the binding variable. This address can be passed around, and used to access the object from elsewhere, but crucially uses only a small amount of space and time to move, rather than the costs for a full object. This can reduce space usage by an exponential amount[1]. In the presence of mutability, of course, objects must be shared to receive each others' updates. Further still, the system could share all terms that happen to be the same, a process known as hash-consing. This would again allow arbitrary space improvements for some programs, but is often quite expensive to perform.

[1]Although it is also possible to use asymptotically more space due to the overhead of pointers, so these models are incomparable[4].

In either case, all of the above space descriptions make a particular assumption: an object stops counting towards space the instant that it does not appear in the term. This is an intuitive property, and one that is commonly adopted for models of space in a functional language, but it is not the only choice that can be made, and it is certainly not the only choice that is made amongst programming languages in use today. One very common choice is to represent all procedure calls in stack frames, for their efficiency on modern hardware. While it is possible to maintain the space expectations above, this is not always done, as there are costs to doing so[2].

For a faithful representation of this model of the lambda calculus, it is important that objects are still collected expediently, even if that comes before the function has finished execution. Furthermore, stack frames must not be created unless necessary for the computation, a requirement called tail-call elimination. That is, a variable is reachable if it is used later in the function, not if it is merely in scope. It should be noted that even reachability is still an approximation, as all garbage collection must be: for instance, a function which is potentially called, but is not defacto called will be kept live, and transitively all of the portions of its closure. This can cause a *space leak*, where-in some data which is actually not needed is artificially kept live much longer than it should be. Although we are presenting a number of optimizations and decisions which affect the amount of space which is used, it should be noted that space leaks can be entirely within the correct behavior for the language, and be introduced by programmers. In fact, the situation may become much worse with mutable references, as now an arbitrarily large amount of data could be kept reachable inside a space-leak. This issue occurs in practice.

Not only are there models with less stringent requirements on collection, systems may go further. Concretely, if it is known that some portion of an object is never used

[2]For instance, the requirement of no space cost to tail calls is incompatible with full stack traces. Early collection is in conflict with the deterministic destruction order guaranteed by languages such as C++ and Rust.

again, that portion may be collected, leaving a dangling pointer to the deleted portion. These have been implemented in several ways. A few researchers have used runtime type inference as a method of collecting untagged data, allowing the collection of data whose type is unconstrained (and thus are unneeded: semantically one could have instantiated it any type, including one without data, without a loss in meaning)[5,7,10]. Very recently, Proust published a dissertation including a mostly static collection[23]. Proust's stronger condition is called *access*, in contrast to reachability. No formal relation has been established yet between these two methods. In general of course, since there are arbitrarily precise analyses on programs, we can always create a stronger, safe space semantics via invocation of those analyses, via any better approximation of usefulness. Even small optimizations may push a language into a stricter class: GHC Haskell has some special cases in the garbage collector for identifying tuples which will immediately be projected from, so that only the relevant portions are kept live[27].

This variability in the space semantics for a language has been noticed by researchers before. In 1998, Clinger defined an order on several space complexity semantics[3], including the stack semantics mentioned. Similarly, Proust instead uses the notion of timeliness, to compare the access condition to reachability. Indeed, Proust's system would lie above the top of Clinger's lattice, called safe-for-space, in reference to the definition of Appel [1]. Note that this is distinct from the safe-for-space property in general, which will be defined.

2.1.1 Garbage Collection

It is worthwhile to briefly discuss the means of implementing garbage collection, and compare the implementability of space semantics.

Tracing garbage collector A tracing garbage collector runs outside of the normal code execution, marking which data is still reachable, and moving or freeing memory for reuse. There are a variety of garbage collection algorithms, with quite different time characteristics. Most tracing garbage collection is "stop-the-world" collection, and pauses the program's execution (possibly multiple threads of execution) while the garbage collector runs.

Reference counting Reference counting is a system where running code will increment a counter when accessing an object, and decrementing that counter when the object is no longer needed. When an object's counter reaches 0, and thus, there are no active references to it, code is run to free that object. Since this mechanism does not ever release cycles of data which point to each other (a so-called reference loop), this is often paired with occasional traced garbage collection. Generally, languages without mutation cannot create reference cycles, so this mechanism is always safe.

Static deletion strategies Several strategies are available which may free objects automatically in code. Syntactic strategies may guarantee deletion at a particular point, or analysis may be performed to estimate object extent. Some examples of the former include C, C++ and Rust. For both C++ and Rust, the existence of destructors may cause the deletion of arbitrarily large quantities of data (or indeed, arbitrary side-effects), and so the timing and order of such destructions is guaranteed. This means that they are always placed at the end of a scope, and as such are not safe for space relative to other methods. For static analysis, some examples are the MLKit compiler for Standard ML[26] , and the Cyclone dialect of C [12].

A *region* is a section of memory which holds many distinct objects, but which is freed all at once. It is desirable for performance, as both allocation and deallocation may be relatively cheap.

Fully static strategies necessarily use more space relative to dynamic strategies for all programs, without additional runtime tests: whether an object is reachable can depend on the result of arbitrary computations. MLKit, in particular, added a traditional garbage collector[9], as data that it was unable to determine the extent for, was otherwise never collected. Proust's method is able to maintain space safety with this sort of strategy by placing some dynamic checks in the code.

2.2 Illustration of space expectations

In this work, the expected space semantics will count the sum space usage of all objects which are reachable from live variables, including the size of stack frames themselves. In order to adequately explain this, several examples will show the expectations, and a possible failure of implementation or optimization which will fail to meet that expectation. In doing so, we treat the size of both numbers and pointers as constant. While this would be exploitable if our integers are unbounded, it will suffice for the examples (and will not affect the proofs). Note also that stack size is included in the space calculation (with one note later in this section).

We will now present a number of code blocks, showing examples of some space complexity expectations, and possible failures, including both mis-optimizations, as well as important representation choices. All code is presented in Standard ML. If the reader is not familiar, a description of needed syntax is presented in the Appendix.

In each block, N is an integer variable whose value is abstract, but used to present program space complexity. The following definition will be useful to construct values of the appropriate size. It constructs an immutable, singly-linked list of size n, filled with the element `0`. Mostly, we will see it called as mkList N, thereby constructing a new value of size N. It is known to be pure.

```
fun mkList n =
   if n <= 0
      (* with n = 0, return the empty list *)
      then nil
      (* else, call mkList recursively to produce
       * the tail of list, and add 0 to the head *)
      else (0 :: mkList (n - 1))
```

In a realistic system, the garbage collector is called infrequently in a program, only when memory is low. Here, we are only interested in the size of the memory that cannot be collected, and allow garbage collection to be performed non-deterministically during code execution. In particular, it is valid (although inefficient) for the garbage collector to run after each evaluation step.

2.2.1 Common sub-expression elimination

Common sub-expression elimination is a program transformation, where the same expression that appears multiple times within a calculation is shared between those calculations. It can be unsafe with respect to space usage when it causes an object's lifetime to be extended too dramatically. In the code below, we will see an example where this extension causes objects which had disjoint lifetimes to be live at the same time, increasing the space complexity of the program.

This code block:

```
fun loop 0 = 0
  | loop count =
   let
      (* mkList count appears here ... *)
      val a = head (mkList count)
      val b = loop (count - 1)
      (* and here *)
      val c = length (mkList count)
   in
      a + c * b
   end

val result = loop N
```

may undergo common sub-expression elimination, to become:

```
fun loop 0 = 0
  | loop count =
   let
      (* mkList count is now computed only once,
       * but is kept in the stack frame while looping *)
      val l = mkList count
      val a = head l
      val b = loop (count - 1)
      val c = length l
   in
      a + c * b
   end

val result = loop N
```

This change causes the program to run in $\Theta(N^2)$ space instead of the $\Theta(N)$ space in the original, since the large `mkList count` must now remain live over the recursive call.

2.2.2 Lambda lifting / Globalization

Lambda lifting (hoisting in [1]) is a program transformation, where a variable inside a lambda body, that does not depend on the value of the argument, may be calculated and shared between calls. Globalization is a subset where constant expressions are transformed into static data.

In the code:

```
(* standard fold-right definition:
 * if the list is empty, return the given value,
 * else, combine with the first element of
 * the list, and recursively call with that
 * element as the new zero value. *)
fun fold (lst, zero, combine) =
   case lst of
      [] => zero
    | (x::xs) => fold (xs, combine x zero, combine)

(* sum over mkList N *)
fun f x = fold (mkList N, x, fn (a, b) => a + b)
```

Lambda lifting may transform `f` to become:

```
val f = let
   val l = mkList N
   (* l is now stored in the closure for f,
    * so it may be reused if f is called
    * multiple times *)
```

```
in
   fn x => fold (l, x, fn (a, b) => a + b)
end
```

This may be space unsafe, as the lifetime of the calculated variable is now longer. In between calls of f, the value of `mkList N` must now be stored, whereas in the un-transformed program, it was only used during the function body's execution. Note though, this pass is only problematic if N is allowed to be a non-constant size. The scope of `f` is necessarily not global if it contains any such non-constant data. In global settings, this is only problematic if data may be of unbounded size. This can occur due to the presence of mutable data, including both mutable references, as included in Standard ML, and call-by-need lazy values, as in Haskell.

In these cases, globalization may be semantically sound, but cause an arbitrarily large object to be stored statically. For objects of statically bounded size, this lifting is safe for space.

The full laziness optimization in the Glasgow Haskell Compiler, as presented in [20] is such a problematic form of lambda lifting, where a definition inside a lambda that does not depend on the argument, is lifted to the outside of that lambda. It has been known to be source of space leaks in Haskell programs[13].

Adding refs, we may produce an example with global scope which is not safe for space:

```
fun f orig =
  let
    val accumRef = ref []
    val () = fold (orig, (),
      fn ((), x) =>
```

```
        accumRef := (x : x : !accumRef))
  in
    !accumRef
  end

val _ = f (mkList N)
```

which may be globalized to the following (since at most one instance accumRef is live at any time):

```
val accumRef = ref []

fun f orig =
  let
    val () = accumRef := []
    val () = fold (orig, (),
      fn ((), x) =>
        accumRef := (x : x : !accumRef))
  in
    !accumRef
  end

val _ = f (mkList N)
```

which causes the list created by `mkList N` to be live in between calls to `f`, which adds space `O(N)` to all other code in the program. In some examples, such as this one, it can be returned to space safety by returning the reference to a small value by the time it had been collected in the program. Thus a further transformation to the following is safe for space:

```
val accumRef = ref []

fun f orig =
  let
    val () = accumRef := []
    val () = fold (orig, (),
      fn ((), x) =>
        accumRef := (x : x : !accumRef))
    val ret = !accumRef
    val () = accumRef := []
  in
    ret
  end

val _ = f (mkList N)
```

This change can be performed whenever the extent of the original object is known and in a location where we can run such code. This is not a property which holds in general, as the same code may appear many distinct times in different runs, and the liveness of the object can differ from run to run based on arbitrary computation and input.

2.2.3 Dead store elimination

In a few cases, eliminating a dead store to a mutable reference may be unsafe: A change which is invisible to the code may be visible to the garbage collector.

An optimizer may notice that one write to `listRef` is unused in the code below:

```
fun refLength listRef =
  length (!listRef)
```

```
fun f 0 = 0
  | f n =
  let
    val listRef = ref (mkList N)
    val i = refLength listRef
    val () = listRef := [] (* dead store *)

    val j = f (n - 1)

    val () = listRef := (mkList N) (* overwriting store *)
    val k = refLength listRef
  in
    i + j + k
  end

val _ = f N
```

and eliminate it to the following code below:

```
fun refLength listRef =
  length (!listRef)

fun f 0 =
  | f n =
  let
    val listRef = ref (mkList N)
    val i = refLen listRef
    (* the dead store is now deleted *)

    val j = f (n - 1)

    val () = listRef := (mkList N)
    val k = refLength listRef
  in
    i + j + k
  end

val _ = f N
```

This change has increased the space usage from $\Theta(N)$ to $\Theta(N^2)$ since the evaluation of `(mkList N)` is now stored in `listRef` during the recursive calls.

2.2.4 Conditionally used variables

Consider the following code block for a recursive function. In one of the two branches, `x` is completely unused. This means that x should not be considered reachable in the second branch, And indeed, leaving x live while recursing in `loop` causes $\Theta(N^2)$ space usage, whereas eliminating it uses only $\Theta(N)$ space.

```
fun loop (count, x) =
      if count = 0
         (* Use x in this branch to calculate
          * the length *)
         then len x
         (* make a non-tail call of loop,
          * using count, but not x *)
         else loop (count-1, mkList N) + 1


val result = loop(N, nil)
```

2.2.5 Closure conversion

When changing from a higher order language to a first-order language (either with function pointers, or with defunctionalization into data-types, as in MLton), one important task is closure conversion, in which an anonymous function is converted to a first-order function, and a data structure of free variables. Closure representation has many apparently valid options, but as Shao and Appel show [1,24], some options may violate space safety (and space efficiency). A linked closure representation, as presented in that work, is one such unsafe option.

Consider the code below. The function `f` captures the list `lst` in its closure, but returns a new function `g`, which does not need the full list. Under a linked closure representation, the closure for `g` will contain a link to the closure for `f`, including the entire original list. This is not safe for space, as `g` would normally only require space for the length of the list, as would happen in a flat closure representation (or the efficient representation of Shao and Appel).

```
let
  val lst = mkList N
  fun f () =
    let
      val len = length lst
      fun g () = len + 1
    in
```

```
      g
    end
in
  g
end
```

2.2.6 Tail-call elimination

An expression is in tail position, if it is the last expression in its branch, which is required to be evaluated before returning a result. For example, in the following code block, the expressions `y + 1`, `bar x`, and even `if x > 0 then y + 1 else bar x` are in tail position.

```
fun f x =
  let
    val y = x * 2
  in
      if x > 0
        then y + 1
      else bar x
  end
```

Tail recursion is the special case where a function calls itself (or another mutually recursive function). We may write the simple `len` function from the appendix as tail-recursive in order to improve performance. With the changes, the function only requires a constant amount of space, and may be compiled into a tight loop using machine registers.

```
fun len xs =
  case xs of
```

```
    [] => 0
  | _ :: xs => 1 + len xs

fun len0 (xs, n) =
  case xs of
    [] => n
  | _ :: xs => len0 (xs, n + 1)
fun len xs = len0 (xs, 0)
```

Tail-call elimination is an implementation technique, where no additional call frame is created for function calls that appear in tail position. Instead, the caller's frame is replaced with the frame for the destination. The destination function then returns, without ever passing through the calling function. Tail-call elimination may asymptotically reduce space usage, and is also beneficial for running time.

This elimination is often quite critical for functional programs: in lieu of mutating loops, functional programming generally requires recursion as the means to repeat computations. In a few cases, direct looping is supported, but it is not the norm for programming.

The following code block demonstrates the effect of tail-call optimizations.

```
fun plus (count, accumulator) =
      if count = 0
         then accumulator
         (* Since we need to do nothing with the result,
          * this recursive call is a tail call, and we
          * no longer need anything from the caller's stack *)
         else plus (count-1, accumulator+1)
   end
```

```
val result = plus (N, N)
```

Without tail-call elimination, each usage of loop will be given a stack frame, using $\Theta(N)$ total space, whereas with the elimination, it will use only $\Theta(1)$.

2.3 Compiler internal representations

In order to optimize programming languages most effectively, it is often beneficial to first translate the source language into a different structure than its syntax tree. This helps convert similar forms into a common language, eliminating features such as implicit evaluation order, or variable mutation. This simplified language helps in performing and understanding operations on code, and the machine used for these optimizations will be defined in terms of the ANF compiler representation.

There are a few common internal representations which are used in compilers:

- Continuation-passing style (CPS)
- A-normal form or monadic normal form (ANF)
- Single static assignment (SSA)
- Direct style

In order to exemplify the representations below, the following code in Standard ML will be converted to a form that looks more similar to the representation. For readability, the infix operators `-`, `*` and `+` will be converted to `sub`, `mul` and `add`. Tuple deconstruction will be represented by a `proj` primitive.

```
fun sqdistance (x, y) (x2, y2) =
  let
    val xdist = x2 - x
    val ydist = y2 - y
    fun square a = a * a
  in
    square xdist + square ydist
  end
```

2.3.1 Direct Style

One of the simplest strategies is to keep the structure of the program, applying only a few de-sugarings. Thus, the function above is mostly unchanged. This has the benefit that implementation for the compiler is simple, the structure of the code is maintained for presentation, and the stack structure of evaluation is kept. On the other hand, the lack of regularization means that code often has to deal with many scoping evaluation order constructs, which would not be required for other forms.

2.3.2 A-normal form

In comparison, one may be more explicit about evaluation order. Instead of using a stack in the course of evaluating simple expressions, expressions are unfolded into definitions, with a name for each individual computation. The call stack is then only used for function calls. There are several ways to do this, but it is worth mentioning A-normal form, as this will be the structure for the machine in this work. Tail calls are explicit: they appear in the last expression of a block.

```
fun sqdistance p1 p2 =
  let
    val x = proj 1 p1
    val y = proj 2 p1
    val x2 = proj 1 p2
    val y2 = proj 2 p2
    val xdist = sub x2 x
    val ydist = sub y2 x
    val square = fn a => mul a a
    val v0 = square xdist
    val v1 = square ydist
  in
    add v0 v1
  end
```

2.3.2.1 Join points

Some authors[14] noticed that A-normal form as originally stated was unable to express some features that other representations (include CPS and SSA) could. In particular, case matches in a non-tail position could not be expressed without code duplication. Join points are an addition of stack-local continuations, in effect, labels, which can be the target of local jumps, without creating an object. It will be worth mentioning only because the language in this work is missing the feature, despite being presented in ANF. The adequacy will be discussed.

2.4 Proof Assistants

Proof assistants are pieces of software which help with the development of formalized proofs, which can then be checked automatically by a trusted and verified kernel. Offering high level commands and automation enables engineering proofs without worrying about many low-level details. Generally, such proofs are written in a high-level proof/programming language, which is eventually compiled down to a simpler format, which can be verified by a small kernel of code. Because proof checking may be done by a very small portion of code, this reduces the amount of manual verification needed to trust a proof, down to the verification of the small kernel, and to checking the correctness of definitions.

The formal proof of these statements offers many benefits over an informal setting. While it is easy to note examples, or otherwise understand issues within the folklore, these often leave boundaries and details quite fuzzy. Formal proof requires specificity, giving very explicit bounds for correctness. The ability to validate a proof's correctness automatically enables refactorability and reusability, affording the option to consider many more varieties than with an informal proof.

Coq[11] is one such proof assistant based on the Calculus of Inductive Constructions, which also supports extraction to OCaml, Haskell and Scheme programs. Unlike some other systems, it is bimodal: there is support for both direct definitions in the language Gallina, as well as the ability to develop proofs in a proof mode. In both cases, this compiles down to the same verifiable core language, but the programmer experience is quite different. Proofs are written interactively as a tree of tactics, manipulating a proof state which consists of many hypotheses, one focused goal, and additional goals to be solved. Tactics will soundly modify the goals or hypotheses, until the proof can be filled in more easily. Several tactics automate proofs, allowing both general proof search, as well as specialized solvers, such as Presburger arithmetic or ring solvers.

Below is some example Coq code, including definitions, functions and proofs. The reader will not need to understand Coq to be able to read this text.

```
Inductive nat : Type :=
| O : nat
| S : nat -> nat.


Definition pred (n: nat): nat :=
  match n with
  | O => O
  | S n => n
  end.

Theorem pred_succ_id : forall n,
  pred (S n) = n.
Proof.
  simpl. reflexivity.
Qed.
```

Chapter 3

Theory

3.1 Syntax

The syntax of our programs will be given in an untyped variant of A-normal form, with booleans, tuples, anonymous functions, and simple I/O.

In some sense, this is meant to simply serve as a middle-ground representation, and avoid complexities with non-local jumps in the graph. We should call out that the simplicity of this language leaves it unable to express some features. There are no mutable data-structures in this language, which are a common feature of unsafe transformations and space leaks. Furthermore, no single statement can construct more than a statically-known amount of data, but this would change if we had added features such as arrays of dynamic length. This means that only a finite amount of data is allocated on any finite sequence of steps. For simplicity, the language does not include recursive functions, but they may be implemented via the Y-combinator, since the language is untyped.

Furthermore, in this language, there is no form of join-point or lightweight continuation mechanism. This would be a desirable item for genuine internal representations; MLton's SXML internal representation uses case-expressions in non-tail positions, and its SSA representation uses blocks with arguments (in lieu of the more common phi-functions).

There are at least two ways to remedy the lack of join points for these programs, both of which suffice. We may create a lambda abstraction for the continuation, which will use space, and then perform a tail-call into it. Since such abstractions will not escape, this simply increases the size used by the corresponding stack frame, by a statically determinable amount. As such, the blowup factor can be computed for a given program, so this representation is safe-for-space. Alternatively, inlining the tail-continuations may be used, which is fully space-efficient (space-non-increasing even) as code size is not counted, but we should convince ourselves that passes will not change the two branches in a disjoint manner. In either case, the lack of such features is not significant for correctness of passes that are blind to them.

Our definitions are given with assumption of the following data:

- *var* is a type
- *addr* is a type
- For any type of store values *sv*, *heap sv* is a type
- *var* and *addr* have decidable equality.

The language will be interpreted by a CESK abstract machine. That is, the state consists of code (an expression), and an environment, store and continuation stack. The language of expressions is in A-normal form, so evaluation order is explicit, and a stack is used only for function calls.

We will take alloc to be a function, where $\text{alloc}(\sigma_1, sv) = (\sigma_2, a)$ means that after allocating a store value sv in the heap σ_1, we have the heap σ_2 and the store value is located at a fresh address a. $\sigma(a)$ looks up the store value at the address a in heap σ. Because the only source of addresses is the alloc function, this is always guaranteed to be present if the address comes from a valid state. There is an empty heap empty, where no addresses are valid.

The steps below should elucidate a few points. Constants (just booleans in our language) never allocate, but tuples and lambda abstractions do. The stack changes only for two expressions: return and the non-tail function application. For a tail call, no new stack frame is created. In this language, there are two forms of events that can occur, reads

Configuration ::= $\langle e, \rho, \sigma, \kappa \rangle$

$$
\begin{array}{lcl}
e \in \text{exp} & ::= & \textbf{let } x = (x, \ldots, x) \textbf{ in } e \\
 & | & \textbf{let } x = x\ x \textbf{ in } e \\
 & | & \textbf{let } x = \pi_i x \textbf{ in } e \\
 & | & \textbf{let } x = \textbf{write } \mathrm{x} \textbf{ in } e \\
 & | & \textbf{let } x = \textbf{read in } e \\
 & | & \textbf{let } x = \mathrm{b} \textbf{ in } e \\
 & | & \textbf{let } x = \textbf{fn } x \Rightarrow e \textbf{ in } e \\
 & | & \textbf{if } x \textbf{ then } e \textbf{ else } e \\
 & | & \textbf{ret } x \\
 & | & x\ x \\
 & & \\
b \in \text{const} & ::= & \text{true} \\
 & | & \text{false} \\
 & & \\
v \in \text{value} & ::= & \textbf{vaddr } \mathrm{a} \\
 & | & \textbf{vconst } \mathrm{b} \\
 & & \\
\rho \in \text{env} & ::= & (x,\ v) :: \rho \\
 & | & \cdot \\
 & & \\
\kappa \in \text{stack} & ::= & (x,\ \rho,\ e) :: \kappa \\
 & | & \cdot \\
 & & \\
sv \in \text{svalue} & ::= & \textbf{sclos } (x,\ e,\ \rho) \\
 & | & \textbf{stuple } (v,\ \ldots,\ v)
\end{array}
$$

$a \in$ addr
$x \in$ var
$\sigma \in$ heap svalue

Figure 3.1: Configuration syntax

and writes, here only a single bit of information. The **write** form will output a boolean which has been computed by the program. The **read** form contains the only non-determinism that can appear in the language, reading may return either true or false, with the appropriate value appearing in the I/O trace with the event.

A machine for a program P will start in the state $\text{start}(P) = \langle\ P,\ \cdot,\ \text{empty},\ \cdot\ \rangle$, and will end when evaluating **ret** x with the empty stack (where x is in the environment).

3.2 Definitions of Space Safety & Efficiency

In order to define when a program is space-efficient with respect to another, we may first want to look at the simpler, symmetric relation of program equivalence. There are many ways to define program equivalence. Perhaps the simplest such relations are those which apply to programs that terminate with simple values. Two programs are equivalent if they terminate with the same value (for an appropriate definition of value sameness). But this definition is often unsuitable for many interesting sorts of programs. For programs that run on a real computer, we likely want them to perform actions other than returning a single value. They may send incremental data to various sources, but also receive data from a user, or other source, and perform different actions based on that data.

Instead, we may wish to consider I/O trace equivalence. An I/O trace for a program is a possibly infinite stream of observable events which have occurred in the program, both the output sent, and input received. Then, two program runs are equivalent precisely when their traces are the same. So we can say that two programs are fully equivalent, if whenever a trace is possible in one, it is possible in the other. This is the baseline we will consider for relative space efficiency of programs. In some cases, we do not need bidirectionality: if our source program always makes progress or finishes, then we need only the forward direction for our language with I/O. Since our language is deterministic up to input, this is true for our language.

$\boxed{\langle e_1, \rho_1, \sigma_1, \kappa_1 \rangle \to \langle e, \rho_2, \sigma_2, \kappa_2 \rangle \ ! \ io}$ $\qquad$ $\boxed{io := \varepsilon | \mathbf{write}\ b | \mathbf{read}\ b}$

$$\frac{\rho(x_1) = v_1,\ \ldots,\ \rho(x_n) = v_n \qquad \mathrm{alloc}(\sigma, \mathbf{stuple}\ (v_1, \ldots, v_n)) = (\sigma', a)}{\langle \mathbf{let}\ x = (x_1, \ldots, x_n)\ \mathbf{in}\ e, \rho, \sigma, \kappa \rangle \to \langle e, (x, \mathbf{vaddr}\ a) :: \rho, \sigma', \kappa \rangle \ ! \ \varepsilon}$$

$$\frac{\rho(x_f) = \mathbf{vaddr}\ a_f \qquad \sigma(a_f) = \mathbf{sclos}\ (x_l, e_l, \rho_l) \qquad \rho(x_a) = v_a}{\langle \mathbf{let}\ x = x_f x_a\ \mathbf{in}\ e, \rho, \sigma, \kappa \rangle \to \langle e_l, (x_l, v_a) :: \rho_l, \sigma, (x, e, \rho) :: \kappa \rangle \ ! \ \varepsilon}$$

$$\frac{\rho(x_t) = \mathbf{vaddr}\ a_t \qquad \sigma(a_t) = \mathbf{stuple}\ (v_1, \ldots, v_n)}{\langle \mathbf{let}\ x = \pi_i x_t\ \mathbf{in}\ e, \rho, \sigma, \kappa \rangle \to \langle e, (x, v_i) :: \rho, \sigma, \kappa \rangle \ ! \ \varepsilon}$$

$$\frac{\rho(x_w) = \mathbf{vconst}\ b}{\langle \mathbf{let}\ x = \mathbf{write}\ x_w\ \mathbf{in}\ e, \rho, \sigma, \kappa \rangle \to \langle e, (x, \mathbf{vconst}\ true) :: \rho, \sigma, \kappa \rangle \ ! \ \mathbf{write}\ b}$$

$$\frac{}{\langle \mathbf{let}\ x = b\ \mathbf{in}\ e, \rho, \sigma, \kappa \rangle \to \langle e, (x, \mathbf{vconst}\ b) :: \rho, \sigma, \kappa \rangle \ ! \ \varepsilon}$$

$$\frac{}{\langle \mathbf{let}\ x = \mathbf{read}\ \mathbf{in}\ e, \rho, \sigma, \kappa \rangle \to \langle e, (x, \mathbf{vconst}\ b) :: \rho, \sigma, \kappa \rangle \ ! \ \mathbf{read}\ b}$$

$$\frac{\mathrm{alloc}(\sigma, \mathbf{sclos}\ (x_l, e_l, \rho)) = (\sigma', a)}{\langle \mathbf{let}\ x = \mathbf{fn}\ x_l \Rightarrow e_l\ \mathbf{in}\ e, \rho, \sigma, \kappa \rangle \to \langle e, (x, \mathbf{vaddr}\ a) :: \rho, \sigma', \kappa \rangle \ ! \ \varepsilon}$$

$$\frac{\rho(x_b) = \mathbf{vconst}\ true}{\langle \mathbf{if}\ x_b\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f, \rho, \sigma, \kappa \rangle \to \langle e_t, \rho, \sigma, \kappa \rangle \ ! \ \varepsilon}$$

$$\frac{\rho(x_b) = \mathbf{vconst}\ false}{\langle \mathbf{if}\ x_b\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f, \rho, \sigma, \kappa \rangle \to \langle e_f, \rho, \sigma, \kappa \rangle \ ! \ \varepsilon}$$

$$\frac{\rho(x_f) = \mathbf{vaddr}\ a_f \qquad \sigma(a_f) = \mathbf{sclos}\ (x_l, e_l, \rho_l) \qquad \rho(x_a) = v_a}{\langle x_f\ x_a, \rho, \sigma, \kappa \rangle \to \langle e_l, (x_l, v_a) :: \rho_l, \sigma, \kappa \rangle \ ! \ \varepsilon}$$

$$\frac{\rho(x_r) = v_r}{\langle \mathbf{ret}\ x_r, \rho, \sigma, (x_l, e_l, \rho_l) :: \kappa \rangle \to \langle e_l, (x_l, v_r) :: \rho_l, \sigma, \kappa \rangle \ ! \ \varepsilon}$$

Figure 3.2: Step Relation

The situation for space-safety is a bit more complicated than for equivalence. Unlike equivalence, it is a directed relationship. That is to say, there are some transformations which are genuine improvements, but of course we would not hope that the reverse transformation, which worsens space (possibly quite strongly) should be related in the same way. Of course we can require that a program use strictly less memory to be space-efficient, but this is quite strict. Instead, the definition will have to be loosened. We will use a (monotonic) function f to scale up the space usage from the source program, showing that this scaled-up usage bounds the space usage of the target instead. We can say that a program P_2 is f-space-efficient with respect to P_1 if the space usage is decreasing after applying the blowup function f. That is, we imagine something of the sort $f(size_1) >= size_2$. For many purposes it will suffice to know that allowing f to be an affine function (that is, of the form $n \mapsto m \cdot n + k$) will agree with the notion of space complexity.

Returning back to the full definitions of space safety and efficiency, we can make a similar progression. If the programs P_1 and P_2 are equivalent in behavior and they always terminate, then certainly P_2 is space-efficient with respect to P_1 if it uses less maximum space for it to execute on every corresponding I/O trace. This solves the situation of accounting for some differences due to I/O, but this is not quite the whole story: First, from a theoretical standpoint, we generally don't want to exclude infinite runs, whose space usage may be bounded or unbounded, especially since they may still be productive. Second, we wish to ensure that memory is used at roughly the same time. An interactive system, such as an operating system or web server may be online for months, or even years. Requiring large amounts of space at one point, should not imply that it is free to use an excess amount of space in the future. So this sort of definition is not sufficient for the guarantees that we want.

One factor is that the states may be reordered. It is perfectly sound, safe, and beneficial to swap two intermediate expressions around for various reasons, so a simple linear

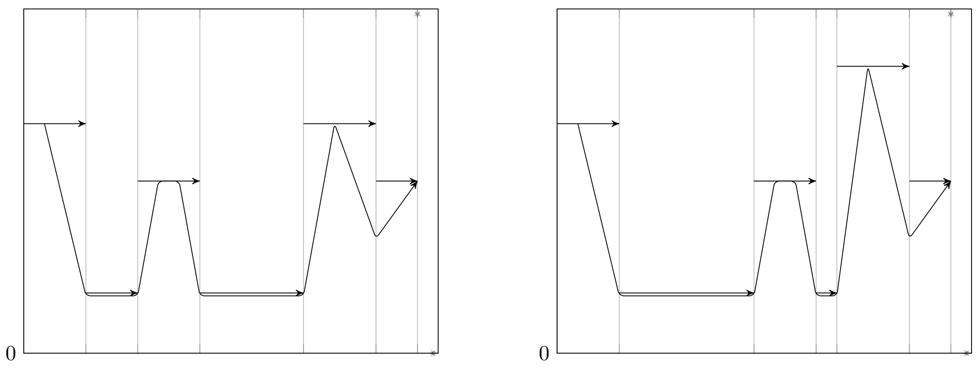


Figure 3.3: Diagram of grouping-based space definition
In this case, the second trace is space efficient with regards to the first, with a multiplicative constant 5/4 (that is, for the function $n \mapsto 5 \cdot n/4$).

grouping won't quite work. One potential definition is to allow us to group up states into blocks: Given the two programs' (potentially infinite) sequence of states, a proof of safety for that sequence consists of picking a finite prefix of each, proving that the maximum space usages are related for those two lists of states, then prove safety for the remaining two sequences. More formally, we have the following co-inductive rules for proving f-efficiency of state sequences, where $|s|$ is the space used by a state s:

$$\text{co}\ \frac{\begin{array}{cc} tr'_s \gtrsim_f tr'_t & tr_s = [s_1, \ldots, s_m] +\!\!+\ tr'_s \\ f(\max_{i=1\ldots m}|s_i|) \geq \max_{j=1\ldots n}|t_j| & tr_t = [t_1, \ldots, t_n] +\!\!+\ tr'_t \end{array}}{tr_s \gtrsim_f tr_t} \qquad \text{co}\ \frac{}{\cdot \gtrsim_f \cdot}$$

This construction forces states to be mostly in order, but allows some states to be grouped together to handle re-orderings. Thus we can imagine such a proof as setting up a sequence of high-water marks throughout the space contour of the program (as in Figure 3.3).

Unfortunately, some passes which perform interleaving re-orderings may now be difficult or impossible to prove space efficient, despite appearing reasonable. If a transformation swaps bindings in a way which allows interleavings, then this definition would require a finite bound on the interleavings, which means that some passes, albeit rather contrived ones, may have no proof of space efficiency; see Appendix B for an example transformation. It is certainly not unbearable to use this definition for proofs, as the example transformation that exhibits it must be rather contrived, but it is worrying, and may indicate unneeded complexity of the criterion, affecting the difficulty of proving space-safety in more reasonable cases.

It also can leave the situation just as bad as full-run space usage: known-terminating programs again have their space usage collapsed into the maximum total usage, as each entire run can be placed into a single group.

In light of I/O trace equivalence, there should be a rather clear way to demarcate consistent times in the two programs. Letting I/O events be the posts unifying the times between programs, efficiency may defined as efficiency for the high-water marks between each two events (and between the start of the program and the first event). This happily fits our criterion for local space checking, but also is realistic for an interactive program: we should be able to assign a proportional amount of memory at each I/O event, and get the same result. While perhaps the granularity of the I/O events in this language is a bit fine, an I/O event often represents a long pause and potential context switching. It would be worthwhile to ensure that space is low during these events.

So more explicitly, between any two corresponding I/O events in the source and target, with finite lists of states between those events $[s_1, \ldots, s_m]$ and $[t_1, \ldots, t_n]$, we have the condition $f(\max_{i=1\ldots m} |s_i|) \geq \max_{j=1\ldots n} |t_j|$ (using our blow-up function f). Continuing on with this sort of definition, we can get one additional simplification. Consider that a sequence of pure steps $s_1, s_2, \ldots, s_m$ and $t_1, t_2, \ldots t_n$ for which we have the condition. With a small bit of algebra we get the following reduction:

$$\boxed{s_1 \to^* s_2 \ ! \ \vec{io}} \qquad \frac{s_1 \to s_2 \ ! \ \varepsilon \qquad s_2 \to^* s_3 \ ! \ \vec{io}}{s_1 \to^* s_3 \ ! \ \vec{io}}$$

$$\frac{s_1 \to s_2 \ ! \ \mathbf{write}\ b \qquad s_2 \to^* s_3 \ ! \ \vec{io}}{s_1 \to^* s_3 \ ! \ \mathbf{write}\ b :: \vec{io}} \qquad \frac{s_1 \to s_2 \ ! \ \mathbf{read}\ b \qquad s_2 \to^* s_3 \ ! \ \vec{io}}{s_1 \to^* s_3 \ ! \ \mathbf{read}\ b :: \vec{io}}$$

Figure 3.4: Multiple steps relation

$$\begin{aligned}
& f(\max_{i=1\ldots m} |s_i|) \geq \max_{j=1\ldots n} |t_j| \\
= \ & \max_{i=1\ldots m} f(|s_i|) \geq \max_{j=1\ldots n} |t_j| \quad \text{By monotonicity of } f \\
= \ & \forall j, \max_{i=1\ldots m} f(|s_i|) \geq |t_j| \\
= \ & \forall j, \exists i, \ f(|s_i|) \geq |t_j|
\end{aligned}$$

So, showing that such a collection of steps is f-efficient is precisely the same as justifying each target state in t with some source state in s, and showing the condition for that pair.

To formally define efficiency, start states, and reachability should be briefly defined:

Definition 1 (f-efficiency of programs)**.** *A target program P_t is f-efficient for a program P_s if:*

For every state t with start(P_t) $\to^*$ t ! $\vec{io}$, there is a state s with start(P_s) $\to^*$ s ! $\vec{io}$, such that: $f(|s|) \geq |t|$

This definition is slightly stronger than the definition regarding the space usage between I/O events. A proof of this definition also requires valid space usage on diverging sequences of steps without I/O, which were not required by ensuring space-safety between any two I/O events. This implies that transformations may not move a large allocation before a potentially infinite loop, which is again, quite appropriate following our assumption that I/O takes a while. We get the concise tagline: "You can't keep a lot of data for a long time", where a lot of data is not known to be in the same space complexity, and a long time is any time not known to be a finite sequence of pure computation. Completeness may be fitted back into this form: (where soundness is already implied by space-safety). a program P_t is a complete representation of a program P_s if for every state s in the source (with start$(P_s) \to^* s \; ! \; \vec{io})$, there is a target state which has start$(P_t) \to^* t \; ! \; \vec{io}$. A program transformation T is complete if $T(P)$ is always a complete representation of P. Since space-safety contains the reverse relation, it already impl

With this in hand, we can define space-safety and efficiency of program transformations.

Definition 2 (Space-safety of transformations)**.** *A transformation T between programs is safe-for-space when:*

For all programs P, there exist constants m and k (that may depend on P), such that

$T(P)$ is $(n \mapsto m \times n + k)$-efficient relative to P.

Definition 3 (Space-efficiency of transformations)**.** *A transformation T between programs is space-efficient when:*

There is a constant m, such that for all programs P, there is a constant k, such that

$T(P)$ is $(n \mapsto m \times n + k)$-efficient relative to P.

There is one detail that sticks out from these definitions: the final value of the program is irrelevant. This is a deliberate choice; it avoids questions of address/store value equivalence, and greatly simplifes the definitions. The behavior of such a return can be easily added onto any particular program, by wrapping the program in a function body, calling that function in a non-tail context, and writing the resulting value out. In other settings, it would not be particularly onerous to add one final state and I/O event for returning the last value, but there seems to be little reason to add that here.

Below are the concrete space semantics for the language defined earlier. The semantics will count the space used by both the program stack and the reachable heap objects. Since our code is of constant size, we choose not to count it. That said, we add a constant amount of space to every object, in accounting for factors such as object metadata for garbage collection. This, plus the size of addresses themselves can crucially change which transformations we are allowed to justify as safe-for-space. It may be noted again, that the size of addresses is constant. This is perhaps slightly odd from a computational point of view, and eliminates some space distinctions which would otherwise be made, (such as the possible improvement of an evaluation which never shares) but ultimately is the most sensible option for our machine definition, and is one taken by other works. Because our heap is garbage collected, making the size of addresses depend on the size of the heap could create circular dependencies. Nevertheless, addresses are not themselves examinable, so there seems to be little risk of inadvertently strengthening our compuations.

3.3 Proving space-safety

Although the definition of the space-safety relation makes clear when we can relate two different states, the question still appears as to how we might find such justifying statesa for a proof, as the steps may not be completely aligned. For a general transformation of programs, just as the steps of the two programs are not perfectly aligned, so too will the heaps not be fully aligned. It is not in general the case that the heap of the source program, and the heap of the target program are the same, even up to an isomorphism of locations.

$\text{size}_{clos}((x_l, c_l, \rho_l)) = |\text{free\_vars } c_l \setminus \{x_l\}|$

$\text{size}_{svalue}(\textbf{stuple } vs) = 1 + |vs|$
$\text{size}_{svalue}(\textbf{sclos } clos) = 1 + \text{size}_{clos}(clos)$

$\text{size}_{stack}([]) = 0$
$\text{size}_{stack}(frame :: \kappa) = 1 + \text{size}_{clos}(frame) + size_{stack}(\kappa)$

$\text{size}_{state}(\langle c, \rho, \sigma, \kappa \rangle) = 1 + |\text{free\_vars } c| + \text{size}_{stack}(\kappa) +$
$\quad \Sigma_{a \in \text{closure}_\sigma(\mathcal{R}_{active}(c,\rho) \cup \mathcal{R}_{stack}(\kappa))} \text{size}_{svalue}(\sigma(a))$
where
$\quad \mathcal{R}_{active}(c, \rho) = \text{addrs}(\rho, \text{free\_vars } c)$
$\quad \mathcal{R}_{stack}([]) = \{\}$
$\quad \mathcal{R}_{stack}((x_r, c, \rho) :: \kappa) = \text{addrs}(\rho, \text{free\_vars } c \setminus \{x_r\}) \cup \mathcal{R}(\kappa)$
$\quad \text{addrs}(\rho, xs) = \{a \mid \exists x \in xs, \rho(x) = \textbf{vaddr } a\}$

$\text{closure}_\sigma(as) = \{a' \in \text{dom}(\sigma) | \exists a \in as, \exists p : a \to_\sigma^* a'\}$
where
$\quad a \to_\sigma a' \iff \sigma(a) = sv$, and $\textbf{vaddr } a \in sv$.
$\quad a \to_\sigma^* a' \iff a = a' \vee (a \to_\sigma a^\dagger \wedge a^\dagger \to_\sigma^* a')$.
$\quad \textbf{vaddr } a \in \textbf{stuple } vs \iff \textbf{vaddr } a \in vs$
$\quad \textbf{vaddr } a \in \textbf{sclos } (x_l, c_l, \rho_l) \iff \exists x, x \text{ free in } c_l \wedge x \neq x_l \wedge \rho_l(x) = \textbf{vaddr } a$

Figure 3.5: Space usage of configuration

To relate the states, we will use a standard inductive binary relation. Thus in general, to prove that two programs are related, we must create remappings between the portions of their associated states, and prove the validity of such. Furthermore, we must create a remapping between the potential live sets of the source, and the potential live sets of the target, and prove their safety. It is this generality, in which we prove space safety for many live sets, which enables a proof by induction of steps.

Consider the following code fragment, and how we might want to address the space safety for a machine that has begun executing it.

```
fun swap tup =
  let
    val (fst, snd) = tup
  in
    (snd, fst)
  end
```

How does the space usage of the program change when we call `swap`? There are precisely two possibilities:

- tup is used here, and then discarded, as it is not live anywhere else: The program's total space is unchanged.
- tup is live elsewhere in the program: The program's total space has increased by the size of a 2-tuple.

In this quite simple example, we have already introduced confusion over the total space of a program. Answering the question of whether that tuple is live, is possible based on program state, but is a rather complex affair. It may be held live through any path through the heap from any live variable in any stack frame. Importantly, those paths may alias, and it is this aliasing that keeps the object live and allows us to share space efficiently. If we are to exactly relate the space of two programs, it is not enough to consider numerical changes throughout an induction (or at the very least, it is quite difficult), but we must keep a full relation on live addresses. If liveness were only checked infrequently, it could possibly be easier to keep alignment, but in general, we want to account for garbage collection to occur at any point in the program, and so liveness must be meticulously tracked. This would require a rather complicated statement about the aliasing of paths.

Instead, we may generalize over the sets which we prove safety with regards to. For doing so, one definition should be made. A set of addresses is *closed* if it is equal to its own closure, i.e., whenever an address is present, it is a valid address, and for any address present in the store value it points to, that address is in the set. There is one very useful property for the collection of closed sets in a given heap. The set of closed sets is inductively generated by the operations of heaps: specifically the construction of an empty heap, and the allocation of a new object. Specifically, we have the following theorems:

- $\text{closed}_{empty}(as)$ iff $as = empty$
- Whenever σ is a valid heap (that is, containing no store values with dangling pointers relative to σ), and sv is a store value,

  Given $(\sigma', a) = \text{alloc}\ (\sigma, sv)$

  a set as is closed in σ' if and only if

  $as \setminus \{a\}$ is closed in σ, and, if $a \in as$ and $\textbf{vaddr}\ a' \in sv$, then $a' \in as$.

A question remains as to how to relate these two sets. Remember that in general the heaps need not be isomorphic, and the machine states which are related can be vastly different. We relate these two sets by means of a relation between individual addresses which is constructed throughout the execution of the program. As we allocate new objects in the programs, we expand the relation to include their addresses. This has the rather convenient property that steps which do not allocate maintain the relation unchanged, and statements about the root-sets may be made independently. For the proofs in this work, the collection of all closed sets will suffice[1], but transformations which take into account more intricate liveness information may likely need to restrict the available sets to those it can prove are possible, in order to be prove safety for them. The key point is that we can "pay" the cost at allocation time, by expanding our property to include the new object.

[1]Some features, such as mutation will force us to expand the set in order to maintain this property. We will have to consider a set of "potentially closed sets", so that mutation is included, but the notes about analyses restricting these sets apply.

Chapter 4

Proofs of transformations

4.1 Lifting / Globalization

It often occurs that a program has expressions which may be computed several times during the execution, but always produce the same value. In these situations, we often wish to lift the expressions up to a higher scope, so that they are not computed as frequently. In doing so, we can improve the program, and also enable additional optimizations. A value which is known to be constant may simplify other code which is determined by its value. In addition, time and space can possibly (but not necessarily) be improved. If an object is moved up in scope from a section of code which is infrequently reachable, to a section which is frequently reachable, this may not be safe for space. Furthermore, careless lifting of objects which are mutable, and may thus have arbitrarily large reachable space may also not be safe for space. In these situations, undesirable failures of space-safety can be avoided through extra analysis and optimization.

```
fun f 0 = 0
  | f n =
  let
    fun getList () =
      if n < 0
        then mkList n
        else []
    val k = f (n - 1)
  in
    length (getList ()) + k
  end
```

can be soundly lifted to:

```
fun f 0 = 0
  | f n =
  let
    val lst = mkList n
    fun getList () =
      if n < 0
        then lst
        else []
    val k = f (n - 1)
  in
    length (getList ()) + k
  end
```

which is not safe for space. The line creating the list: `val lst = mkList n` was never going to be executed in the source program. But, the variable `lst` still appears syntactically, so the list that it points to will not be garbage collected. This adds an overhead of \`Theta(n)` space to each stack-frame for `f`. In total, this brings the space usage for `f` from $\Theta(n)$ to $\Theta(n^2)$. This issue was significant partially because `n` was not constant, but the issue is still sufficient to block space-efficiency for constant-sized data. There are enough constant costs present that we will be able to justify a multiplicative factor of k, where k is the constant size of the lifted data. Of course, k can be arbitrarily large, so this will not allow us to have space-efficiency, without artificial limits.

As such, we will consider lifting objects up to the top-level scope of the code, which is only run once. This should be simple, and increase space usage by at most the total size of the newly lifted data, as instead of numerous (but possibly zero) copies of a datum being allocated at various points in the program, exactly one copy is allocated. Unfortunately, it is not the case that the additive factor is the only increase in the language as defined. In defining the operational semantics of the language we made the common choice to include all referenced variables in the closures of lambda abstractions and of stack frames. Moreover, values which are stored in closures and stack frames always count towards the size of these objects. Since a stack frame may have size 1, if it is empty, and size $(1+k)$ when k definitions are added to its environment, the space usage may in general be multiplied by one plus the number of terms which was lifted to the global scope.

Without arbitrary limits, attempts to globalize all applicable data will not meet the definition of space efficiency. In comparison to actual compilers, this comes as a bit of a shock. For a real compiler, globals will not need to be captured in any closure. Instead, operating systems provide the ability to load data at a known point in memory, which can be referenced directly from code. In order to have a space-efficient transformation, the costs of this model will need to be adjusted to account for static locations, so that we represent proper globalization. Nevertheless, the cost of the objects themselves is still counted, although it will be merely an additive constant.

Definition 4 (Global lifting)**.**

Let binds be a list of (variable, definition) pairs, where a definition is either a constant b, a tuple $(x_1, \ldots, x_n)$ or an abstraction $\mathbf{fn}\ x_l \Rightarrow e_l$.

The program transformation is defined by the following pseudocode, where the syntax of the language is separated by backticks from the meta-syntax:

```
globalize(defs, P) = bind_globals defs (elim_defs (map fst defs) P)
  where
  elim_defs def_vars c =
    match c with
    | ‘let‘ x ‘=‘ ‘fn‘ xl ‘=>‘ cl ‘in‘ cr =>
      if member x def_vars
        then elim_defs def_vars er
        else ‘let‘ x ‘=‘ ‘fn‘ xl ‘=>‘ (elim_defs def_vars cl) ‘in‘ (elim_defs def_vars cr)
    | ‘let‘ x ‘=‘ c1 ‘in‘ cr =>
      if member x def_vars
        then elim_defs def_vars er
        else ‘let‘ x ‘=‘ c1 ‘in‘ (elim_defs def_vars cr)
    | ‘if‘ xb ‘then‘ ct ‘else‘ cf‘ =>
      ‘if‘ xb ‘then‘ (elim_defs def_vars ct)
              ‘else‘ (elim_defs def_vars cf)
    | _ => c
    end
  bind_globals defs c =
    match defs with
    | [] => c
    | (x, d) :: defs’ =>
      bind_globals defs’ (‘let‘ x ‘=‘ d ‘in‘ c)
    end
```

This definition does not include the analysis for choosing definitions to move. Instead there is a syntactic proposition which is used for the proof.

Theorem 1 (Lifting is safe-for-space (fully verified))**.** *Global lifting of a list of immutable objects $defs$ is safe for space, with multiplicative constant $|defs|$, and as an additive constant, the total size of the globals*

Theorem 2 (Globalization is space-efficient (fully verified))**.** *Global lifting of a list of immutable objects $defs$ is space-efficient, with an additive constant equal to the total size of the globals (and a multiplicative constant of 1)*

For brevity, we will refer to the original program as the source program, and the transformed program as the target program, and likewise use these adjectives for comparing any two portions of the states.

As the two programs move forward, they remain relatively synchronized with each other, but not exactly. We can imagine the pair of source and target states to be in one of three stages. First, we may be binding globals in the target, which are not bound in the source. Second, we may be performing bindings in the source, which are skipped in the target (since they were already performed), and third, we may take a step on both. Conceptually, we may have a sort of automaton between these stages, where a step can be consumed from either the source, target or both. The choice of which step to take is based on details about the code (such as whether a binding has been eliminated), but this will help modularize the proof into segments.

Here ϵ indicates that no step is taken, whereas $\rightarrow$ indicates a step taken. On each edge we have a pair $\langle s, t\rangle$ of progress on the source and target programs.

The proof of space-efficiency will be performed by induction over the steps in the target run. That is, for every step we take in the target program, we will perform some steps in the source program and show that that new state is related to the target state. First, we will pass over each global binding: We will take 0 steps in the source program for each step binding a global in the target program. Afterwards, for each step in the target,

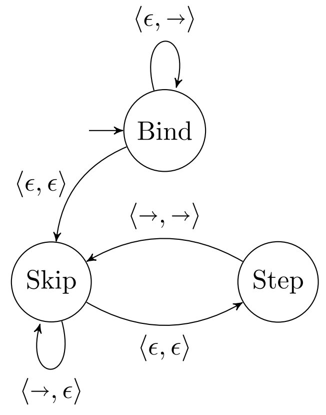

$\langle \epsilon, \rightarrow \rangle$
Bind
$\langle \epsilon, \epsilon \rangle$
$\langle \rightarrow, \rightarrow \rangle$
Skip
Step
$\langle \epsilon, \epsilon \rangle$
$\langle \rightarrow, \epsilon \rangle$

Figure 4.1: Stage automaton for globalization

we will skip over 0 or more source steps (corresponding to globals, which were eliminated in the target), then take one step in each, of the same sort. Because we may take an arbitrary number of steps in the source program as we skip definitions which have been moved in the target program, this offers some complications of mechanization as to proving termination. The dual situation appears in proofs of completeness, for binding the globals. Thus we will split the above automaton into three small proofs, with one outermost theorem switching between the appropriate states. Below, each of the proofs are shown in a different color, with black lines indicating control passing via the outermost theorem. This again does not include all information about the stages and transitions, but should help organize our proofs.

Our inductive hypotheses will need to vary throughout the progression through the various stages, to ensure that the relationships are sufficiently detailed to make further progress. The data of our relations will be the definition data, an environment ρ_{defs} of finished definitions, and a current position in the globals. As such the state Bind above is really indexed by a natural number. In addition, a relationship between addresses in the source and target will be constructed as the proof progresses forward through steps. Whenever we allocate the "same" object in both heaps, the two addresses will be added to the relation. Each address appears in the source program at most once (and we will treat it

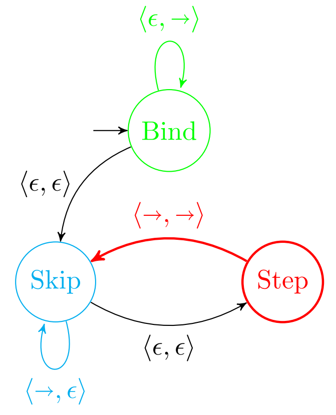

$\langle \epsilon, \rightarrow \rangle$
Bind
$\langle \epsilon, \epsilon \rangle$
$\langle \rightarrow, \rightarrow \rangle$
Skip
Step
$\langle \epsilon, \epsilon \rangle$
$\langle \rightarrow, \epsilon \rangle$

Figure 4.2: Stage automaton for globalization (colored)

as a function of that side). In the target program, there may be multiple incoming edges, as any binding which has been lifted will be allocated only once in the target, but zero or more times in the source. This relation will help to maintain the induction hypothesis on size of closed address sets, since the image can be directly computed. It also keeps the relations from being mutually inductive.

For example, when taking steps in the source and target:

$$\langle \textbf{let } x = (y, z) \textbf{ in } c_1, \rho_1, \sigma_1, \kappa_1 \rangle \rightarrow \langle c_1, (x, \textbf{vaddr } a_1) :: \rho_1, \sigma_1', \kappa_1 \rangle$$

and

$$\langle \textbf{let } x = (y, z) \textbf{ in } c_2, \rho_2, \sigma_2, \kappa_2 \rangle \rightarrow \langle c_2, (x, \textbf{vaddr } a_2) :: \rho_2, \sigma_2', \kappa_2 \rangle$$

the pair (a_1, a_2) will be added to the relation.

The relations between different data are shown below. They are indexed by the two lists of definitions, as well as the address-relation, when necessary. There are four indices present in these relations, namely: $defs$, ρ_{defs}, r and n. Defs is the list of definitions, and ρ_{defs} an environment matching those global variables to corresponding values. The variable r is the relation between addresses which we are building up over time. The variable n

is perhaps less obvious. Here, it marks how many globals still need to be processed. It will be 0 throughout most of the program. In general, this relation consists mostly of two separate conditions: First each item in the source has a related item present in the target, and second, all global items (in ρ_d) are present according to their definition in $defs$.

It is worth calling out the environment relation. It may seem as though the requirements for specific variables to be present in one of the two sets is sometimes abit too strict, but these requirements are fairly tight. Initially, we might assume that all globals are live in the target environment. Of course, this is not true, because abstractions which appear in the globals will have only a prefix of this environment. Not only the globals themselves will have this property, but any lambda abstraction which appears in the body of such a globalized lambda, will also have an environment which does not contain all globals.

vconst $b \approx_r$ **vconst** b
vaddr $a_1 \approx_r$ **vaddr** a_2 if $(a_1, a_2) \in r$

$(\rho_1, xs_1) \approx_{r,\rho_{defs}} (\rho_2, xs_2)$
if:
1. Whenever $x \in xs_1$, and $\rho_1(x) = v_1$, and $\rho_{defs}(x) = v_2$, then $v_1 \approx_r v_2$
2. Whenever $x \in xs_1$, and $x \in xs_2$, and $\rho_1(x) = v_1$, there exists v_2 with $\rho_2(x) = v_2$, and $v_1 \approx_r v_2$
3. Whenever $x \in xs_2$, and $\rho_{defs}(x) = v$ then $\rho_2(x) = v$

$c_1 \approx^{defs,n} c_2$
if:
1. bind_globals(take n $defs$, elim_defs (map fst $defs$) c_1) $= c_2$
2. defs_agree($defs, c_1$)

$(x, \rho_1, c_1) \approx^{defs}_{r,\rho_{defs}} (x, \rho_2, c_2)$
if:
1. $c_1 \approx^{defs,0} c_2$,
2. $x \notin \rho_{defs}$
3. $(\rho_1, \text{free\_vars}(c_1) \setminus \{x\}) \approx_{r,\rho_{defs}} (\rho_2, \text{free\_vars}(c_2) \setminus \{x\})$

stuple $vs_1 \approx^{defs}_{r,\rho_{defs}}$ **stuple** vs_2
if vs_1 and vs_2 have the same length and for each i, $vs_1(i) \approx_r vs_2(i)$
sclos $clos_1 \approx^{defs}_{r,\rho_{defs}}$ **sclos** $clos_2$
if $clos_1 \approx^{defs}_{r,\rho_{defs}} clos_2$

$\sigma_1 \approx_{r,\rho_{defs}} \sigma_2$
if:
1. Whenever $\sigma_1(a_1) = sv_1$, there exists a_2 and sv_2 with $\sigma_2(a_2) = sv_2$, $(a_1, a_2) \in r$, and $sv_1 \approx_{r,\rho_{defs}} sv_2$
2. Whenever $\rho_{defs}(x) =$ **vaddr** a, there is some sv with $\sigma_2(a) = sv$ and defined_svalue($defs(x), sv$)

$\kappa_1 \approx^{defs}_{r,\rho_{defs}} \kappa_2$
if κ_1 and κ_2 have the same length and for each i, $\kappa_1(i) \approx^{defs}_{r,\rho_{defs}} \kappa_2(i)$

$\langle c_1, \rho_1, \sigma_1, \kappa_1 \rangle \approx^{defs,n}_{r,\rho_{defs}} \langle c_2, \rho_2, \sigma_2, \kappa_2 \rangle$
if:
1. Whenever $(x, v) \in \rho_{defs}$, we have defined_value($defs(x), v$)
2. Whenever **vaddr** $a \in \rho_{defs}$, $a \in dom(\sigma_2)$
3. map fst $\rho_{defs} =$ map fst (drop n $defs$)
4. Whenever $(a_1, a_2) \in r$, $a_1 \in dom(\sigma_1)$ and $a_2 \in dom(\sigma_2)$
5. $c_1 \approx^{defs,n} c_2$
6. $\rho_1, \approx^{\text{free\_vars}(c_2)}_{r,\rho_{defs}} \rho_2$
7. $\sigma_1 \approx_{r,\rho_{defs}} \sigma_2$
8. $\kappa_1 \approx^{defs}_{r,\rho_{defs}} \kappa_2$

Figure 4.3: Relations for Lifting/Globalization

defined_value : definition $\to$ value $\to$ Prop
defined_value$(b, \textbf{vconst } b)$
defined_value$((x_1, \ldots, x_n), \textbf{vaddr } a)$
defined_value$(\textbf{fn } x_l \Rightarrow c_l, \textbf{vaddr } a)$

defined_svalue : env $\to$ definition $\to$ value $\to$ Prop
defined_svalue$(\rho_{defs}, (x_1, \ldots, x_n), \textbf{stuple } (v_1, \ldots, v_n))$
 if each $v_i = \rho_{defs}(x_i)$
defined_svalue$(\rho_{defs}, \textbf{fn } x_l \Rightarrow c_l, \textbf{sclos } (x_l, c_l, \rho_l))$
 if for each x in free_vars$(c_l \backslash \{x_l\})$,
there is some v such that $\rho_{defs}(x) = \rho_l(x) = v$.

```
defs_agree : list (var * definition) -> exp -> Prop
defs_agree(defs, c) =
  match c with
  | ‘let‘ x ‘=‘ xs ‘in‘ cr =>
     match defs(x) with
     | Some xs’ => xs = xs’ /\ defs_agree(defs, cr)
     | Some _ => False
     | None => defs_agree(defs, cr)
     end
  | ‘let‘ x ‘=‘ b ‘in‘ cr =>
     match search v defs with
     | Some b’ => b = b’ /\ defs_agree(defs, cr)
     | Some _ => False
     | None => defs_agree(defs, cr)
     end
  | ‘let‘ x ‘=‘ ‘fn‘ xl ‘=>‘ cl ‘in‘ cr =>
    match search v defs with
    | Some (fn xl’ => cl) => xl’ not in defs /\ xl = xl’
        /\ elim_defs (map fst defs) cl = cl’
        /\ defs_agree(defs, cl) /\ defs_agree(defs, cr)
    | Some _ => False
    | None => vl not in defs /\ defs_agree(defs, cl)
        /\ defs_agree(defs, cr)
    end
  | ‘let‘ x ‘=‘ _ ‘in‘ cr =>
    if x in defs then False else defs_agree(defs, cr)
  | ‘if‘ xb ‘then‘ ct ‘else‘ cf =>
    defs_agree(defs, ct) /\ defs_agree(defs, cf)
  | ‘ret‘ x => True
  end
```

Figure 4.4: Auxilliary definitions for Lifting/Globalization

```
same_head : exp → exp → Prop
same_head(‘let‘ x ‘=‘ RHS ‘in‘ cr_1, ‘let‘ x ‘=‘ RHS ‘in‘ cr_2)
same_head(‘if‘ x ‘then‘ cr_t1 ‘else‘ cr_f1,
‘if‘ x ‘then‘ cr_t2 ‘else‘ cr_f2)
same_head(‘tail‘ x_f x_a, ‘tail‘ x_f x_a)
same_head(‘ret‘ x, ‘ret‘ x)

head_global : list var → exp → Prop
head_global(defVars, ‘let‘ x ‘=‘ _‘in‘ _) = x ∈ defVars
```

Figure 4.5: Auxilliary definitions for Lifting/Globalization (2)

We will begin proving the statement for initial global bindings. Throughout this section, we will make the implicit assumption that the definition variables are unique and are well-scoped. That is, each variable appears at most once as the variable for a definition. Each variable that appears in a definition appears in a previous portion of the definitions list.

The following lemma is useful for handling the changes in ρ_{defs} affecting old values.

Lemma 1 (defined_svalue Extension). *If defined_svalue*$_\rho(sv, d)$ *and* $x \notin dom(\rho)$*, then defined_svalue*$_{(x,v)::\rho}(sv, d)$

Theorem 3 (Initial bind related). *Suppose start_state*$(P) \approx^{defs,n+1}_{\cdot,\rho_g} \langle c_g, \rho_g, \sigma_g, \kappa_g\rangle$ *and* $defs(n) = (x_b, d)$*, and* $\langle c_g, \rho_g, \sigma_g, \kappa_g\rangle \to \langle c'_g, \rho'_g \sigma'_g, \kappa'_g\rangle$*, and map fst* ρ_g *= map fst drop* $(n+1)$ $defs$ *Then start_state*$(P) \approx^{defs,n+1}_{\cdot,\rho_g} \langle c'_g, \rho'_g, \sigma'_g, \cdot'\rangle$

Proof. Several portions of the relation are trivial. The source environment and heap are empty, as well as the address relation and both stacks. This leaves only conditions 1, 2, 3, 5, 6, and 7

The code relation, condition 5 is simple in each case. By the definition of bind_-globals, c_g is of the form **let** $x = d$ **in** c_r, and it is easy to see for each case that $c'_g = c_r$. The environment relation (6) is likewise simple. The source environment is empty, and during the iteration, $\rho_{defs} = \rho'_g$, so the global condition can be handled by the identity. Condition 3, that each global variable (in ρ_{defs}) is in the heap, holds simply as well. Since the only addresses that are introduced come from the heap allocation function, and the new heap is immediately used, it can be shown by induction on steps that every address which appears in a state points to an object in the heap, so this holds for ρ'_g. Condition 2 requires that the n parameter which is used for code also describes the globals. This holds by noting that $x_b :: \text{drop } (n+1) \text{ (map fst } defs) = \text{drop } n \text{ (map fst } defs)$ when $defs(n) = (x_b, d)$.

The only interesting remaining portions are conditions 1 and 7.

By cases on d:

- When d is a constant b, we have:
  $\rho'_g = (x_b, \mathbf{vconst}\ b) :: \rho_g$,
  $\sigma'_g = \sigma_g$, and $\kappa'_g = \kappa_g = \cdot$ by the unique step for b.

  1. We want to prove that any $(x, v) \in \rho'_g$ has a corresponding agreeing definition. Knowing that $\rho'_g = (x_b, \mathbf{vconst}\ b) :: \rho_g$, we have the two cases, $(x, v) = (x_b, \mathbf{vconst}\ b)$, or $(x, v) \in \rho_g$. Both cases are immediate, the latter by the precondition

  7. The heap is unchanged, so all we need to do is show that each svalue is related to ρ'_g. This follows by the lemma above (along with uniqueness of bindings).

- When d is a tuple/abstraction, we have:
  $\rho'_g = (x_b, \mathbf{vaddr}\ a) :: \rho_g$, for some a,
  $\kappa'_g = \kappa_g = \cdot$ by each corresponding step. σ'_g is some new heap with one extension.

  1. We want to prove that any $(x, v) \in \rho'_g$ has a corresponding agreeing definition. Knowing that $\rho'_g = (x_b, \mathbf{vaddr}\ a) :: \rho_g$, we have the two cases, $(x, v) =$

$(x_b, \textbf{vaddr}\ a)$, or $(x, v) \in \rho_g$. The latter case is by the precondition. The former case is also trivial, here defined_value($\textbf{vaddr}\ a, d$) holds for both sorts.

7. Since we've allocated a new object, we will need to establish that it corresponds to its definition. Existing entries will use the lemma.

   – For $d = xs$: We allocated **stuple** vs, where $vs(i) = \rho_g(xs(i))$. It is enough to define it in the old environment, and use the extension lemma. The values are related by the lookup lemma.

   – For $d = \textbf{fn}\ x_l \Rightarrow c_l$: We allocated **sclos** (x_l, c_l, ρ_g). Use the extension lemma. In order to show defined_svalue$_{\rho_g}$(**sclos** $(x_l, c_l, \rho_l), \textbf{fn}\ x_l \Rightarrow c_l$), we must show that for every free variable x besides x_l, there is a value v which is present in both ρ_g and $\rho_l = \rho_g$ (that is, we must have $\rho_g(x) = v$).

     Since the definitions are well-scoped, the subset starting at n (that is, starting with $d = \textbf{fn}\ x_l \Rightarrow c_l$). is well-scoped. Thus, for each free variable in c_l (besides x_l), there is a corresponding entry in the remaining defs (i.e. drop $(n + 1)$ $defs$). Via condition 4 of the precondition, the variables in drop $(n + 1)$ $defs$) are the same as those in $\rho_{defs} = \rho_g$. So each variable is in ρ_g.

□

In addition to relatedness, we will have a space-safety condition for a program state. The brunt of difficulty is with establishing the size of heap data, and this is all that this relation ensures. The other components of size (primarily stack size) can be established just via relatedness.

This space safety can now be established for the initial steps. For the initial state start_state(P), the initial heap is empty, so in particular, any closed set as of addresses is also empty. Thus, for $m = \Sigma_{a \in \text{addrs}(\rho_{defs})} \text{size}_{svalue}(\sigma_2(a))$, and $n = \Sigma_{a \in \text{addrs}(\rho_{defs})} \text{size}_{svalue \setminus defs}(\sigma_2(a))$,

$\langle c_1, \rho_1, \sigma_1, \kappa_1 \rangle \;\; \geq^{m,n}_{r,\rho_{defs}} \;\; \langle c_2, \rho_2, \sigma_2, \kappa_2 \rangle$ If for every set as_1 closed in σ_1,

- $(1 + |defs|) \cdot \Sigma_{a \in as_1} \text{size}_{svalue}(\sigma_1(a)) + m \geq$ $\Sigma_{a \in r_*(as_1) \cup \text{addrs}(\rho_{defs})} \text{size}_{svalue}(\sigma_2(a))$
- $\Sigma_{a \in as_1} \text{size}_{svalue}(\sigma_1(a)) + n \geq$ $\Sigma_{a \in r_*(as_1) \cup \text{addrs}(\rho_{defs})} \text{size}_{svalue \setminus defs}(\sigma_2(a))$

where
$\text{addrs}(\cdot) = \varnothing$
$\text{addrs}((x, \textbf{vconst}\ b) :: \rho) = \text{addrs}(\rho)$
$\text{addrs}((x, \textbf{vaddr}\ a) :: \rho) = \{a\} \cup \text{addrs}(\rho)$
$r_*(as)$ is the image of as under the map $a \mapsto r(a)$.
$size_{svalue \setminus defs}$ is the size of the svalue, except ignoring variables in $defs$ when calculating the size of closures.

Figure 4.6: Space safety relation for lifting/globalization

we always have start_state$(P) \geq^{m,n}_{\cdot,\rho_{defs}} s_2$ when start_state$(P) \approx^{defs,k}_{r,\rho_{defs}} s_2$. Note that this holds not just for the entire sequence, but for any partial sequence through the initial definitions.

Leaving skipping for later, we want to ensure that relatedness and space-safety are preserved by taking the same step in both the source and target program. For doing so, it is helpful to establish lemmas, passing relatedness from component to component.

Lemma 2 (Environment lookup related). *If $x \in xs_1$, and $(\rho_1, xs_1) \approx_{r,\rho_{defs}} (\rho_2, xs_2)$, then $\rho_1(x) \approx_r \rho_2(x)$*

Lemma 3 (Environment extension related). *If $x \notin \rho_{defs}$, Then if $(\rho_1, xs_1) \approx_{r,\rho_{defs}} (\rho_2, xs_2)$, and $v_1 \approx_r v_2$, $((x, v_1) :: \rho_1, xs_1) \approx_{r,\rho_{defs}} ((x, v_2) :: \rho_2, xs_2)$,*

Lemma 4 (Environment subset related). *If $x \notin \rho_{defs}$, Then if $(\rho_1, xs_1) \approx_{r,\rho_{defs}} (\rho_2, xs_2)$, $xs_1' \subset xs_1$, and $xs_2' \subset xs_2$, then $(\rho_1, xs_1') \approx_{r,\rho_{defs}} (\rho_2, xs_2')$,*

Lemma 5 (Heap lookup related). *If $(a_1, a_2) \in r$, and $\sigma_1 \approx_{r,\rho_{defs}} \sigma_2$
then $\sigma_1(x) \approx_r \sigma_2(x)$*

Lemma 6 (Value related after alloc)**.** *If v_1 is valid in σ (no dangling addresses), and $a_1 \notin dom(\sigma)$, then if $v_1 \approx_r v_2$, $v_1 \approx_{(a_1,a_2)::r} v_2$*

Lemma 7 (Environment related after alloc)**.** *If ρ is valid in σ, and $a_1 \notin dom(\sigma)$, then if $(\rho_1, xs_1) \approx_r (\rho_1, xs_2)$, $(\rho_1, xs_1) \approx_{(a_1,a_2)::r} (\rho_2, xs_2)$*

Lemma 8 (Store value related after alloc)**.** *If sv_1 is valid in σ, and $a_1 \notin dom(\sigma)$, then if $sv_1 \approx_r sv_2$, $sv_1 \approx_{(a_1,a_2)::r} sv_2$*

Proof. Using both the environment and value lemmas above for **sclos** and **stuple** respectively □

Lemma 9 (Heap related after alloc)**.** *If σ_1 and σ_2 are valid heaps (containing no dangling addresses), and $\sigma_1 \approx_{r,\rho_{defs}} \sigma_2$, and $sv_1 \approx^{defs}_{r,\rho_{defs}} sv_2$, then if $alloc(\sigma_1, sv_1) = (\sigma_1', a_1)$ and $alloc(\sigma_2, sv_2) = (\sigma_2', a_2)$, $\sigma_1' \approx_{(a_1,a_2)::r,\rho_{defs}} \sigma_2'$*

Proof.

1. For each address a by cases on $a = a_1$ or $a \neq a_1$, applying the previous lemma.
2. Globals are immediate since lookups are preserved after allocation, and defined_svalue$_{\rho_{defs}}$ is independent of r.

□

Finally, we can address proving the step portion. Because we need relatedness as well as the new relation, both relatedness and safety are proved together, but safety is needed only for that portion. Keeping the space portion contained can allow this proof to also be used for completeness, where no space-relation is involved. First we will establish the preservation of relatedness, then the two cases for space can be assembled via a lemma.

Theorem 4 (Step related)**.** *Suppose $\langle c_1, \rho_1, \sigma_1, \kappa_1\rangle \approx^{defs,0}_{r,\rho_{defs}} \langle c_2, \rho_2, \sigma_2, \kappa_2\rangle$ and c_1 and c_2 have the same head,*

then if $\langle c_1, \rho_1, \sigma_1, \kappa_1\rangle \to s_1' \ ! \ i$ and $\langle c_2, \rho_2, \sigma_2, \kappa_2\rangle \to s_2' \ ! \ i$

There exists some r', such that $s_1' \approx^{defs,0}_{r',\rho_{defs}} s_2'$ and if $s_1 \geq^{m,n}_r s_2$, then $s_1' \geq^{m,n}_r$

Proof. Some portions are invariant in this section of execution, namely conditions 1 and 2 relating $defs$ and ρ_{defs}, since $defs$ and ρ_{defs} never change. The safety relation will also be unchanged whenever the heap and r are unchanged, so it will be included only for allocations.

By cases on the head of c_1 and c_2:

- **let** $x = (x_1, \ldots, x_n)$ **in** $cr_{\{1,2\}}$:

  We have alloc$(\sigma_1, \textbf{stuple}\ (v_1^1, \ldots, v_1^n)) = (\sigma_1', aa_1)$

  and alloc$(\sigma_2, \textbf{stuple}\ (v_2^1, \ldots, v_2^n)) = (\sigma_2', a_2)$

  so $s_1' = \langle cr_1, (x, \textbf{vaddr}\ a_1) :: \rho_1, \sigma_1', \kappa_1 \rangle$

  and $s_2' = \langle cr_2, (x, \textbf{vaddr}\ a_2) :: \rho_2, \sigma_2', \kappa_2 \rangle$

  where each v_i^j comes from the appopriate environment

  r' will be $(a_1, a_2) :: r$.

  3. Since validity is preserved by heap extension
  4. Immediate by checking equality with a_1 and a_2 and by heap extension.
  5. Unchanged
  6. Immediate by sub-expression
  7. By applying the lemmas for environments. We can use the new allocation and new value lemmas to prove relateness for $(\rho_1', \text{free\_vars}(\textbf{let}\ x = (x_1, \ldots, x_n)\textbf{in}\ cr_1) \cup x)$ and the corresponding entry, then use the subset lemma for the environments, since contains free_vars(cr_1).
  8. By applying the lemma for heaps with new allocations
  9. By applying the lemma for stacks with new allocations.

For safety, we'll describe the lemma below, simply note that the size of the tuples are identical, since their lengths are the same since the construction is syntactically identical.

- **let** $x = x_f x_a$ **in** $cr_{\{1,2\}}$:

  By our previous lemmas, the closures looked up for x_f in the source and target are related. Likewise the values for x_a are related by lookup. Since there are no allocations, we don't need to adjust r.

  3. The heap is unchanged
  4. The heap and r are unchanged
  5. Immediate by sub-expression
  6. By applying the lemmas for subsets and the new variable. The value added was related by lookup as x_a.
  7. The heap is unchanged
  8. All previous frames are maintained. For the new closure, the code and variable conditions hold by the fact that the binder x is not a global variable. The environment follows simply by the subset lemma.

- **let** $x = \pi_i x_t$ **in** $cr_{\{1,2\}}$:

  The values at x_t are related by environment lookup. These are addresses since since we took a step. The tuples fetched via those addresses are relatd by heap lookup. The values projected from those tuples are related by the definition of relation on **stuple** .

  r will be unchanged.

  3. The heap is unchanged
  4. The heap and r are unchanged
  5. Immediate by sub-expression

6. By applying the lemmas for subsets and the new variable.
7. The heap is unchanged
8. The stack is unchanged.

- **let** $x = b$ **in** $cr_{\{1,2\}}$:

  b is syntactically the same constant in both programs, so it is related.

  r will be unchanged.

  3. The heap is unchanged
  4. The heap and r are unchanged
  5. Immediate by sub-expression
  6. By applying the lemmas for subsets and the new variable.
  7. The heap is unchanged
  8. The stack is unchanged.

- **let** $x =$ **write** x_w **in** $cr_{\{1,2\}}$:

  The lemma definition enforced that the I/O taken is the same, so these must be related. A small side lemma will ensure the values at x_w will be related enough to ensure that.

  The result of writing is the constant true, so it is related to itself.

  r will be unchanged.

  3. The heap is unchanged
  4. The heap and r are unchanged
  5. Immediate by sub-expression
  6. By applying the lemmas for subsets and the new variable.
  7. The heap is unchanged

8. The stack is unchanged.

- **let** $x =$ **read** b **in** $cr_{\{1,2\}}$:

  Since the I/O events are the same, the value read in here is the same. Thus the result is related.

  r will be unchanged.

  3. The heap is unchanged
  4. The heap and r are unchanged
  5. Immediate by sub-expression
  6. By applying the lemmas for subsets and the new variable.
  7. The heap is unchanged
  8. The stack is unchanged.

- **let** $x =$ **fn** $x_l \Rightarrow cl_{\{1,2\}}$ **in** $cr_{\{1,2\}}$:

  First we will need to establish that the new closures are related.

  1. Trivial by reduction as before
  2. By defs_agree, the variable x_l is not a global
  3. By the subset lemma

  Thus the closures **sclos** (x_l, ρ_1, cl_1) and **sclos** (x_l, ρ_2, cl_2) are related for r.

  r' will be $(a_1, a_2) :: r$, where the closures were allocated at a_1 and a_2 respectively.

  3. $a_1 \in dom(\sigma')$ since they were returned by allocation. Old values are still valid by heap extension.
  4. As with the tuple, by checking equality with a_1 and a_2 and by heap extension.
  5. Immediate by sub-expression

6. By applying the lemmas for subsets and the new variable.

7. By applying the heap allocation lemma

8. By the stack allocation lemma

For safety, we need to determine the relative sizes of the closures. We have that elim_defs (map fst $defs$) $(c_1) = c_2$, so it follows by a simple induction that free_vars$(c_2) \subset$ free_vars$(c_1) \cup$ (map fst $defs$).

Thus, for the lifted and the globalized cases:

$$\begin{aligned}
& size_{svalue}(\textbf{sclos } (x_l, \rho_2, cl_2)) \\
= \quad & 1 + |\text{free\_vars}(cl_2) \backslash \{x_l\}| \\
\leq \quad & 1 + |(\text{free\_vars}(cl_1) \cup (\text{map fst } defs)) \backslash \{x_l\}| \\
= \quad & 1 + |\text{free\_vars}(cl_1) \backslash \{x_l\} \cup (\text{map fst } defs)| \quad \text{since } x_l \notin \text{map fst } defs \text{ by defs\_agree)} \\
\leq \quad & 1 + |(\text{free\_vars}(cl_1) \backslash \{x_l\}| + |\text{map fst } defs| \\
\leq \quad & 1 + |(\text{free\_vars}(cl_1) \backslash \{x_l\}| + |defs| \\
= \quad & |defs| + size_{svalue}(\textbf{sclos } (x_l, \rho_1, cl_1)) \\
\leq \quad & (1 + |defs|) \cdot size_{svalue}(\textbf{sclos } (x_l, \rho_1, cl_1))
\end{aligned}$$

$$\begin{aligned}
& size_{svalue \backslash defs}(\textbf{sclos } (x_l, \rho_2, cl_2)) \\
= \quad & 1 + |\text{free\_vars}(cl_2) \backslash \{x_l\} \backslash (\text{map fst } defs)| \\
\leq \quad & 1 + |(\text{free\_vars}(cl_1) \cup (\text{map fst } defs)) \backslash \{x_l\} \backslash (\text{map fst } defs)| \\
= \quad & 1 + |\text{free\_vars}(cl_1) \backslash \{x_l\} \backslash (\text{map fst } defs)| \\
\leq \quad & 1 + |\text{free\_vars}(cl_1) \backslash \{x_l\}| \\
= \quad & size_{svalue}(\textbf{sclos } (x_l, \rho_1, cl_1))
\end{aligned}$$

□

To preserve the space relation, we'll want to make a general lemma. This lemma will work for both notions of space. We should also note, that whenever as is closed in σ', for alloc$(\sigma, sv) = (\sigma', a)$, then $as \backslash \{a\}$ is closed in σ, so the induction hypothesis will be able to be applied.

Lemma 10 (Space of allocation)**.** *Let sz_1 and sz_2 be any two size functions for store values (i.e. from store values to natural numbers)*

Suppose we have that:

1. $alloc(\sigma_1, sv_1) = (\sigma'_1, a_1)$
2. $alloc(\sigma_2, sv_2) = (\sigma'_2, a_2)$
3. *For all a', $(a', a_2) \notin r$*
4. $a_2 \notin as_{defs}$

and there are constants $k \geq 1$, and $n \geq 0$, such that

$$k \cdot space_{sz_1,\sigma_1}(as \backslash \{a_1\}) + n \geq space_{sz_2,\sigma_2}(r_*(as \backslash \{a_1\}) \cup as_{defs})$$

and

$$k \cdot sz_1\ sv_1 \geq sz_2\ sv2$$

then we have:

$$k \cdot space_{sz_1,\sigma'_1}(as) + n \geq space_{sz_2,\sigma'_2}(((a_1, a_2) :: r)_*(as) \cup as_{defs})$$

where

$space_{size,\sigma}(as) = \Sigma_{a \in as} size_{svalue}(\sigma(a))$ *(using 0 when $\sigma(a)$ not defined)*

Proof. We will want to use the fact that $as\backslash\{a_1\}$ is closed in the old heap, so that we can establish its relation on size using the precondition. With that, there are two cases for whether the new address a is present in as or is not. Since lookups are preserved by heap extensions, if $a_1 \notin as$, then the precondition is sufficient.

From the third condition above, we know that $\{a_2\}$ is disjoint from $r_*(as)$, since otherwise a pair (a', a_2) would be present in r. Then we also know that for any $a \in r_*(as)$, $\sigma_2(a) = \sigma_2'(a)$, since said a is not equal to a_2. For the same reason we know the same for every $a \in as_{defs}$. From the first condition above, we know that $\sigma_1(a) = \sigma_1'(a)$ for any $a \in as$. Since $a_1 \notin as\backslash\{a\}$, we have that $((a_1, a_2) :: r)_*(as\backslash\{a\}) = r_*(as\backslash a)$

For brevity, we'll use the syntax $|as|_\sigma$ for $space_{size,\sigma}(as)$, and $|a|$ for $space_{size,\sigma}(\{a\})$.

Then $|as|_{\sigma_1'} = |a_1|_{\sigma_1'} + |as_0|_{\sigma_1'} = |a|_{\sigma_1'} + |as_0|_{\sigma_1}$ The image $r_*'(as) = r_*'(\{a_1\}) \cup r_*'(as_0) = \{a_2\} \uplus r_*(as_0)$, since images distribute over unions, plus the above statements. So we have $|r_*'(as)| = |a_2|_{\sigma_2'} + |as_0|_{\sigma_2}$.

It is easy to show that $|as_1|_\sigma + |as_2|_\sigma \geq |as_1 \cup as_2|_\sigma$ for any set of addresses, so we only need to show:

$$k \cdot |as|_{\sigma_1'} \geq |((a_1, a_2) :: r)_*(as)|_{\sigma_2'} + |as_d efs|_{\sigma_2'}$$

which we can then expand using the above facts to:

$$k \cdot |as\backslash\{a\}|_{\sigma_1} + k \cdot size(sv_1) + n \geq |r_*(as)|_{\sigma_2} + size(sv_2) + |as_d efs|_{\sigma_2}$$

Which is just the sum of the two precondition inequalities.

□

In order to handle writes, we'll need to ensure we can take an appropriate step. This can be proven via the relation, but it is easiest to prove it in the situation that the source program is never stuck, which we'll require elsewhere. This is not strictly needed until the final theorem, but is paired with the Related step above.

Lemma 11 (Can step)**.** *If the source program is never stuck, then if* $s_1 \approx^{defs,0}_{r',\rho_{defs}} s_2$*,* s_1 *and* s_2 *have the same head, and* $s_2 \to s_2' \; ! \; i$*, then there is some* s_1' *with* $s_1 \to s_1' \; ! \; i$*.*

Proof. It is easy to see that the step relation is deterministic whenever $i = \epsilon$, so the only interesting cases are **write** and **read**. For **read** it is immediate, since we can take a step for either of the two possibilites by definition. For **write**, we can show that the two values retrieved from the related environments are related. Since s_2 took a step, it must be some constant. Since they are related constants they are equal, so s_1 takes a step with the same output. □

The final of the three sections will be the proof for skipping. We'll again prove preservation for one skipping step. But we'll also need to prove that we can take as many steps as we need (since it will need to be obviously terminating for a formalized proof). Like before, two lemmas will help reuse some work between the different branches.

It is useful to require the condition that the source program is never stuck for this section. While it is not strictly necessary, it prevents a situation where the source program has a line which will cause the program to become stuck, but that definition is globalized, and the ordering of the globals causes it to not be stuck in the globalized program. If this case were handled, then the condition could be removed; in effect it should require that the definitions are well-scoped in their appearance in the original program in the order they appear, but this requires a much more complicated syntactic definition. The non-stuckness property serves as a simple proxy for this condition, and would not be terribly difficult to replace. Since many of the programs we are interested in compiling have come from a strongly-typed source language, they all have the property that they are not stuck. It is thus a simple property to require.

Lemma 12 (Environment extended in source)**.** *Suppose* $(x, v_g) \in \rho_{defs}$*, and* $v \approx_r v_g$*. Then if* $\rho_1 \approx^{xs}_{r,\rho_{defs}} \rho_2$*, We have* $(x, v) :: \rho_1 \approx^{xs}_{r,\rho_{defs}} \rho_2$

Lemma 13 (Heap extended in source)**.** *Suppose* $(x, \textbf{vaddr}\ a_g) \in \rho_{defs}$*, and alloc*$(\sigma_1, sv) = (\sigma_1', a)$ *and* $\sigma_2(a_g) = sv_g$ *and* $sv \approx^{defs}_{r,\rho defs} sv_g$ *and each of* sv *and* σ *are valid in* σ *Then if* $\sigma_1 \approx^{defs}_{r,\rho_{defs}} \sigma_2$*, We have* $\sigma_1' \approx^{defs}_{(a,a_g)::r,\rho_{defs}} \sigma_2$

Theorem 5 (Skip step)**.** *If* $s_1 \approx^{defs,0}_{r',\rho_{defs}} s_2$*, and* s_1 *is never stuck and is valid, and* s_1*'s head is global (that is, it is skipped in the target), then there exists* s_1' *and* r'*, such that* $s_1 \to s_1'\ !\ \varepsilon$ *and* $s_1' \approx^{defs,0}_{r',\rho_{defs}} s_2$ *and if* $s_1 \geq^{m,n} s_2$*, then* $s_1' \geq^{m,n} s_2$

Proof. The definitions agree by the code relation, so we will have nontrivial cases for each possible global definition: tuples, constants and abstractions. In each case, defs_agree will enforce that it is the correct definition.

- The code is **let** $x = (x_1, \ldots, x_n)$ **in** c_r

  Since the program is not stuck, we have a step to another state $s_1' = \langle c_r, (v, \textbf{vaddr}\ a) :: \rho, \sigma', \kappa\rangle$,

  where $s_1 = \langle \textbf{let}\ x = (x_1, \ldots, x_n)\ \textbf{in}\ c_r, \rho, \sigma, \kappa\rangle$

  Since x is in the globals, $\rho_{defs}(x)$ is defined by condition 2.

  Using condition 1 from the relation, $\rho_{defs}(x)$ is an address $\textbf{vaddr}\ a_g$.

  We will have $r' = (a, a_g) :: r$, and the state be s_1' for the above.

  For relatedness: Again conditions 1 and 2 are constant.

  1. ρ_{defs} is still valid in σ' by heap extension
  2. r is still valid in σ' by heap extension
  3. The code relation is immediate
  4. By the environment source extension lemma above combined with the prior environment allocation lemma
  5. Using the heap source extension lemma above. The tuple $(v_1, \ldots, v_n)$ is related to the heap value pointwise: By defined_svalue, we have that sv_g is the tuple

obtained by searching each of the variables in $(x_1, \ldots, x_n)$ in ρ_{defs}. By condition 2 in the environment, whenever a variable appears in both the source environment ρ and ρ_{defs}, the associated values are related. Thus since each component is related, the tuples are related.

6. By stack relation after allocation

For soundness: Any closed set as in the source is strictly larger than another set as for which the size relation holds. But the image of $\{a\}$ is just $\{ag\}$, which is already present for the relation, since the image is unioned with the addresses in ρ_{defs}. So the condition holds by transitivity.

- The code is **let** $x =$ **fn** $x_l \Rightarrow c_l$ **in** c_r

  Since the program is not stuck, we have a step to another state $s_1' = \langle c_r, (v, \textbf{vaddr}\ a) :: \rho, \sigma', \kappa \rangle$,

  Since x is in the globals, $\rho_{defs}(x)$ is defined by condition 2.

  Using condition 1 from the relation, $\rho_{defs}(x)$ is the an address **vaddr** a_g.

  We will have $r' = (a, a_g) :: r$, and the state be s_1' for the above.

  For relatedness: Again conditions 1 and 2 are constant.

  1. r is still valid in σ' by heap extension

  2. The code relation is immediate

  3. By the environment source extension lemma above combined with the prior environment allocation lemma

  4. Using the heap source extension lemma above. Using defs_agree and defined_-svalue we have that both closures have the same x_l and c_l, so only the environment is nontrivial, that is, we must show $\rho_1 \approx_{r,\rho defs}^{\text{free\_vars(elim\_defs (map fst } defs\text{) } c_l) \setminus \{x_l\}} \rho_l$.

     In this case ρ_l will generally a be a sub-environment.

     Condition 1 is already given by the relation between ρ_1 and ρ_2, it is unchangeed.

   (a) Use the data of defined_svalue: any free variable in c_l is present in ρ_{defs} (and ρ_l). The relation between ρ_1 and ρ_2 can establish value relatedness via condition 2.

   (b) As the other condition, but instead we only need show that the var-value pair is ρ_l.

5. By stack relation after allocation

For space-safety, see the tuple case, it is identical.

- The code is **let** $x = b$ **in** c_r

  Since the program is never stuck, have a step to another state $s_1' = \langle c_r, (v, \textbf{vconst } b) :: \rho, \sigma', \kappa \rangle$,

  Since x is in the globals, $\rho_{defs}(x)$ is defined by condition 2.

  Using condition 1 from the relation, $\rho_{defs}(x)$ is the constant **vconst** b syntactically.

  We will have $r' = r$, and the state be s_1' for the above.

  For relatedness: Again conditions 1 and 2 are constant.

  1. ρ_{defs} and the heap are unchanged
  2. r and the heap are unchanged
  3. The code relation is immediate
  4. By the environment source extension lemma above
  5. The heap is unchanged.
  6. By the environment source extension lemma above

$\square$

Finally, these steps must be combined in a few glue lemmas until we get the final theorems. We will combine the three components, as well as proving the true statements about space from the individual components.

The fact that we can take many steps is necessary as a lemma, particularly for termination purposes. For the formal proof, all recursive definitions must be obviously terminating; this is usually via a shrinking condition, all terminating calls are on a smaller term. In this case, we will use the recursive structure of expressions to ensure termination, although it does make the proof a tad bit more awkward.

Lemma 14 (Skip source steps)**.** *If $s_1 \approx^{defs,0}_{r',\rho_{defs}} s_2$, and s_1 is never stuck and is valid, then there exists r' and s_1', such that s_1' and s_2 have the same head, and $s_1 \rightarrow^* s_1' \ ! \ \varepsilon$, and $s_1' \approx^{defs,0}_{r',\rho_{defs}} s_2$, and if $s_1 \geq^{m,n} s_2$, then $s_1' \geq^{m,n} s_2$*

Proof. By induction on the code, generalizing all other components of the state s_1, as well as the relation r.

Again the only cases which will occur are the three globalizable statements. In the case where a statement is not globalizable, or the variable is not global, we can use s_1 and are done. It follows that s_1 has the same head as s_2 by code-relatedness, since the head of elim_defs will be unchanged if it is not a global. All other conditions are trivial.

The three cases will be entirely similar. For such a global variable, we will note that the expression after a single step changes from **let** $x = RHS$ **in** c_r to c_r. Thus we will use the Skip source step theorem above to generate such a step and show preservation of the relations. Validity and non-stuckness are always preserved by steps. We will get a new relation r', and some state $\langle c_r, \rho, \sigma, \kappa \rangle$. We can verify that the code is c_r by inversion on the resulting step. Apply the induction hypothesis with r', ρ, σ and κ to get a new relation $r^\dagger$ and a new state $s_1^d agger$ with a sequence of steps $s_1 \rightarrow^* s_1^d agger \ ! \ \varepsilon$ and the corresponding relations. Then, we let our r' be this $r^\dagger$, and s_1' be $s_1^d agger$. Composing the single step from the Skip source step along with the sequence of steps from induction will give us the appropriate steps, and since neither component of the composition had I/O, neither do these.

$\square$

The last two components will be a mildly stronger version of the final definition, then the relation between the safe and relatedness conditions and the final statements about space. At this point we ought to define the expected space usage for some definitions.

$$\text{size}_{definition}(b) = 0$$
$$\text{size}_{definition}(\textbf{stuple}\ xs) = 1 + |xs|$$
$$\text{size}_{definition}(\textbf{sclos}\ (x_l, c_l, \rho_l)) = 1 + |\text{free\_vars}(c_l)\backslash\{x_l\}|$$

$$\text{size}_{definition\backslash defs}(b) = 0$$
$$\text{size}_{definition\backslash defs}(\textbf{stuple}\ xs) = 1 + |xs|$$
$$\text{size}_{definition\backslash defs}(\textbf{sclos}\ (x_l, c_l, \rho_l)) = 1 + |\text{free\_vars}(c_l)\backslash\{x_l\}\backslash\text{map fst } defs|$$

Lemma 15 (Globalize safe help)**.** *Suppose that $defs$ agrees with a source program P, and P is syntactically closed, and never stuck.*

for each target state t with steps $start\_state(globalize(defs, P)) \to^* t\ !\ \hat{i}$, such that one of the following holds:

- *(Initial steps)*

  There exists a constant $remaining \leq |defs|$ such that,

  - $start\_state(P) \approx^{defs,remaining}_{\cdot,\rho_t} t$
  - $\hat{i} = \epsilon$
  - $\kappa_t = \cdot$
  - $\Sigma_{\textbf{vaddr}\ a\in\rho_g} size_{svalue}(\sigma_t(a)) = \Sigma_{d\in drop\ remaining\ defs} size_{definition}(d)$
  - $\Sigma_{\textbf{vaddr}\ a\in\rho_g} size_{svalue\backslash defs}(\sigma_t(a)) = \Sigma_{d\in drop\ remaining\ defs} size_{definition\backslash defs}(d)$

  where $t = \langle c_t, \rho_t, \sigma_t, \kappa_t\rangle$

- *(Rest of program) There exists a relation r, and a state s, such that*
  - $start\_state(P) \to^* s \ ! \ \hat{\imath}$
  - $s \approx^{defs,0}_{r,\rho_t} t$
  - $s \geq^{m',n'}_{r,\rho_{defs}} t$

  where

  $m = \Sigma_{d \in defs} size_{definition}(d)$

  $n = \Sigma_{d \in defs} size_{definition \setminus defs}(d)$

Proof. By induction on the steps start_state(globalize($defs, P$)) $\to^* t \ ! \ \hat{\imath}$

In the case that there are no steps, choose the first option (Initial steps): $m' = n' = 0$. $\rho_{defs} = \cdot$. $remaining = |defs|$.
All the conditions are simple.

In the case that there is some step $t' \to t^\dagger \ ! \ i'$ and some previous steps $t \to^* t' \ ! \ \hat{\imath}$. By our induction hypothesis we have one of the two cases above for t'. We want to show it for $t^\dagger$.

- If we are in the initial steps, and remaining > 0:

  We have for the index *remaining* that $defs(remaining) = (x_b, d)$ for some x_b, d. This is sufficient to simplify the take $(remaining + 1)$ $defs$ in the relation to be take $remaining$ $defs + +[(x_b, d)]$. By the definition of bind_globals, and the code relation, we now have that the head of our code in the target is of the form **let** $x_b = d$**in** c_r.

  Note that we have that $\rho'_t = \rho_{defs}$, and $defs$ and ρ_{defs} are connected in the state relation. Since the definitions are well-scoped by assumption, this means that we can successfully take a step. Applying the initial step theorem above will allow us to preserve relatedness and space-safety.

The result will then be in the initial steps, with $(remaining - 1)$ defs remaining.

The adjusted equality on sizes holds by manipulating lists; while it wasn't necessary until now, it is practically easy to add to the lemma for initial steps.

- If we are in the initial steps, and $remaining = 0$, or we are in the rest of the program: These two cases are similar (note that when $remaining = 0$ it is a special case of the other branch). m' and n' are unchanged (and should have achieved m and n). We have validity and non-stuckness by passing these over steps.

  In any case, it is a simple application. First skip steps in the source, then apply related step. There are no complications.

$\square$

Theorem 6. *(Relations imply safety) Suppose s_1 and s_2 are valid states, and $s_1 \approx^{defs,0}_{r,\rho_{defs}} s_2$ and $s_1 \geq^{m,n}_{r,\rho_{defs}} s_2$,*

Then $(1+|defs|)\cdot size_{state}(s_1)+m \geq size_{state}(s_2)$ and $size_{state}(s_1)+n \geq size_{state\setminus defs}(s_2)$. (where $size_{state\setminus defs}$ is as before: vars in $defs$ are not counted in the size of closures, for the heap or stack).

Proof. We will show three separate inequalities for the heap, stack and code/environment.

For the code/environment, we can follow the steps for the closure again (in related step), with the constant cost 1 coming from the state size, and ignoring the $\{x_l\}$ component. For the stack, proceed by induction, applying the fact for closures.

As for the heap, the primary effort is in ensuring that the right-hand address set of our safety condition covers all addresses in the target. We will have that the union of the image of the live addresses in s_1 and the globals (as in our definition for the safety relation) is a superset of the union of live addresses in s_2. Then we apply the condition to get the size relation. This is given by two separate facts:

- Every root in s_2 is a root in s_1 or is global
- Every address in the closure (in σ_2) of the image of a set as from σ_1 is in the globals, or is in the image of the closure of as (in σ_1)

Summing the three portions together will suffice for both.

□

Theorem 7 (Lifting/Globalization are safe-for-space/efficient). *if P is a closed program (containing no free variables), and P is never stuck,*

Then for any state t with $start\_state(globalize(defs, P)) \to^* t\ !\ \hat{i}$, there is a state s with $start\_state(P) \to^* s\ !\ \hat{i}$, such that

$(1 + |defs|) \cdot size_{state}(s) + m \geq size_{state}(t)$

and $size_{state}(s) + n \geq size_{state \setminus defs}(t)$,

where

$m = \Sigma_{d \in defs} size_{definition}(d)$

$n = \Sigma_{d \in defs} size_{definition \setminus defs}(d)$

Proof. By applying the help lemma above, we have two cases:

- Initial steps:

  The source program has an empty set of roots (and thus an empty set of live data), so it has size 0. Thus we must prove in both cases that the expected size is greater than or equal to the actual size; This is fairly trivial, keeping in mind that the set of global addresses is closed, and so equal to its own closure. The free variables must be similarly counted, in the lifting case (in the globalization case they have no cost).

- Rest of program: By Theorem 6.

□

4.1.1 Discussion of mechanization

The above proof as represented in Coq consists of 742 lines of specification, and 2619 lines of proof. The base machine syntax, and several proofs about it, such as lemmas about fresh addresses, validity, non-stuckness, and facts about heaps total to around 720 lines of specification and 760 lines of proof, in a few files. Including comments, the entire development consists of just around 6000 lines of Coq. It is fully axiom-free and is parametrized over the variable type and heap (which have models). It uses a small portion of coq-std++ from the Iris project[21,22], primarily for definitions and tools for sets, as well as a few generic tactics. Overall, the structure of the formalized proof is parallel to the proof above.

It is hopefully worthwhile to discuss a few representation decisions which were beneficial or detrimental to the proof effort, as well as general experiences.

The use of lists throughout for environments and definitions required quite a few lemmas to translate between one and the other; this was exacerbated slightly by the existence of two equivalent propositions for list membership. Even after automation, this added several repetitive steps, especially to fit very particular forms of the relations, such as in the environment relation. For example, we often have a statement like `In x (map fst defs)`, from which we would like to get the pair `(x, d)` with `In (x, d) defs`. This can be done via a lemma, and is not particularly onerous, but is very frequently required in the proof, and is not handled well by automation. Similar issues arise in the heap; here the lookup is a function (written with `!!` in infix position) which returns either `Some sv` or `None`. The forms of `h !! a = Some sv`, and `h !! a <> None` (where <> means not-equals) were cumbersome to translate between. It would have been ideal to use a more standard representation for these properties.

The properties of the closure operation were able to be succinctly described via an induction principle, and this enabled the proofs of every other necessary property about closures, as well as enabling some use in the actual proof (specifically in commuting images with closures). This is an improvement over using equivalence to an actual inductive datatype, since it avoids the translations back and forth. Specifically, the closure of a set of addresses is inductively generated by two operations: any address in the original set is in the closure, and for any address a in the closure, if it points to some sv with a' in sv, then a' is in the closure. Formally this comes out to the following definitions: with two functions to include elements, and an induction principle (which is proof-irrelevant over the actual element-proof, a property which generally holds by parametrization).

```
Parameter closure : `{Addresses sv} -> heap sv -> addrs -> addrs.
Parameter closure_inject : forall h a addrs1,
    elem_of a addrs1 ->
    h !! a <> None ->
    elem_of a (closure _ h addrs1).
Parameter closure_descend : forall h a1 addrs1 v a2,
    elem_of a1 (closure _ h addrs1) ->
    h !! a1 = Some v ->
    addr_in v a2 ->
    h !! a2 <> None ->
    elem_of a2 (closure _ h addrs1).

(* We need to make P dependent on the element proof to
   get Coq to figure out how to generalize,
   but in all cases we force P to be proof-irrelevant with it *)
Parameter closure_ind :
  forall {h: heap sv} (valid: heap_valid h) {addrs1: addrs},
  forall (P: forall (a: addr), elem_of a (closure _ h addrs1) -> Prop)
```

```
(P_inject: forall a, elem_of a addrs1 -> h !! a <> None ->
  forall el, P a el)
(P_descend: forall a v el, P a el -> h !! a = Some v ->
  forall a2 el2, addr_in v a2 -> P a2 el2),
forall a (el: elem_of a (closure _ h addrs1)), P a el.
```

In Coq, this definition may be used via `induction x, H using (closure_ind H1)`, where `H : elem_of x (closure h addrs1)` and `H1 : heap_valid h`. From this we can prove all other principles of interest: the closure agrees with a propositional form of closedness (the closure is closed, and the closure of a closed set is equivalent to the original), as well as properties with unions and images.

The unfolding infrastructure present in std++, along with some tactics for destructing products and existentials were incredibly useful in handling many of the facts about sets, while also enabling additional simplifications.

There were occasional complications due to interactions between various restrictions in Coq. The index $\hat{i}$ on the multiple step relation is not an index in the formalization. This is because having it be so complicates the type-checking for simple functions, such as concatenation. Getting them to type-check would likely require writing these definitions in proof mode, but since we want to prove some facts about them, this is not ideal, since terms created in proof mode are often rather ugly to examine. So instead, the I/O actions are computed from the sequence of steps. Since the list of I/O actions is a list, which is in Set, steps would also have to be in Set (which is already reasonable, since it is *not* irrelevant). Although much of the impact disappeared, this had pushed earlier proofs to require some notionally propositional data be in Set. In general, the difficulty of reasoning about proof terms seemed to force the definitions to be more monolithic.

4.2 Closure Conversion

The efficiency of closure conversion has not been formally proven in this work. Regardless, it is instructive to walk through a the main details for such a proof, so that we can see how to apply the methodologies more easily to other transformations.

Closure conversion is a representation pass in which lambda abstractions in the source code are converted to concrete tuples over their environment and function pointers, as an actual piece of data. That is, if a lambda abstraction uses variables x, y, and z from its environment, we will instead convert it to contain an actual tuple, containing x, y, and z. This is an important step in the concretion of high-level features. The exact structure for this language depends on particular features. In this case, it is made easier by having no direct recursion, but perhaps more difficult by having only single-argument functions

Mechanically, closure conversion is in part far simpler and far more complex than the globalization. The diagram below shows the stages, each only a single step which occur for the program. Unlike globalization, the space used is in lockstep whenever the programs are in the baseline Step stage. This simplifies the situation, although it is still beneficial to quantify over live sets to avoid tracking aliasing and reachability for steps that do nothing but destruct data (and thus can make some data become dead). Nevertheless, the proof will need to be indexed over the stage, and will need to keep some other addresses, like globalization did with its addresses. Furthermore, there is no longer a direct function from a single address to a single address. The converted closures may now take up 3 separate objects on the heap, and this will need to be accounted for in the relations.

Seeing where we've had to add code, we have the following automaton. Since we never deleted anything from the source, every step is either in both the source and the target, or in just the target.

Now, we will want a relation that is indexed by the current stage. As we progress through the code, we will expect to make a corresponding transition in our relation. When

```
let $f = $ fn $x_l \Rightarrow c_l$ in
...
let $y = fx$ in
...
$fx$
```

```
let $f_0 = $ fn $a \Rightarrow$
   let $f_e = \pi_0 a$ in
   let $x_l = \pi_1 a$ in
   let $x_1 = \pi_0 f_e$ in
   let $x_2 = \pi_1 f_e$ in
   ...
   let $x_n = \pi_n f_e$ in
   $c_l'$ in
let $f_e = (x_1, x_2, \ldots x_n)$ in
let $f = (f_0, f_e)$in
...
let $f_0 = \pi_0 f$ in
let $f_e = \pi_1 f$ in
let $a = (f_e, x)$ in
let $y = f_0\, a$ in
...
let $f_0 = \pi_0 f$ in
let $f_e = \pi_1 f$ in
let $a = (f_e, x)$ in
$f_0\, a$
```

Figure 4.7: Example code change for closure conversion

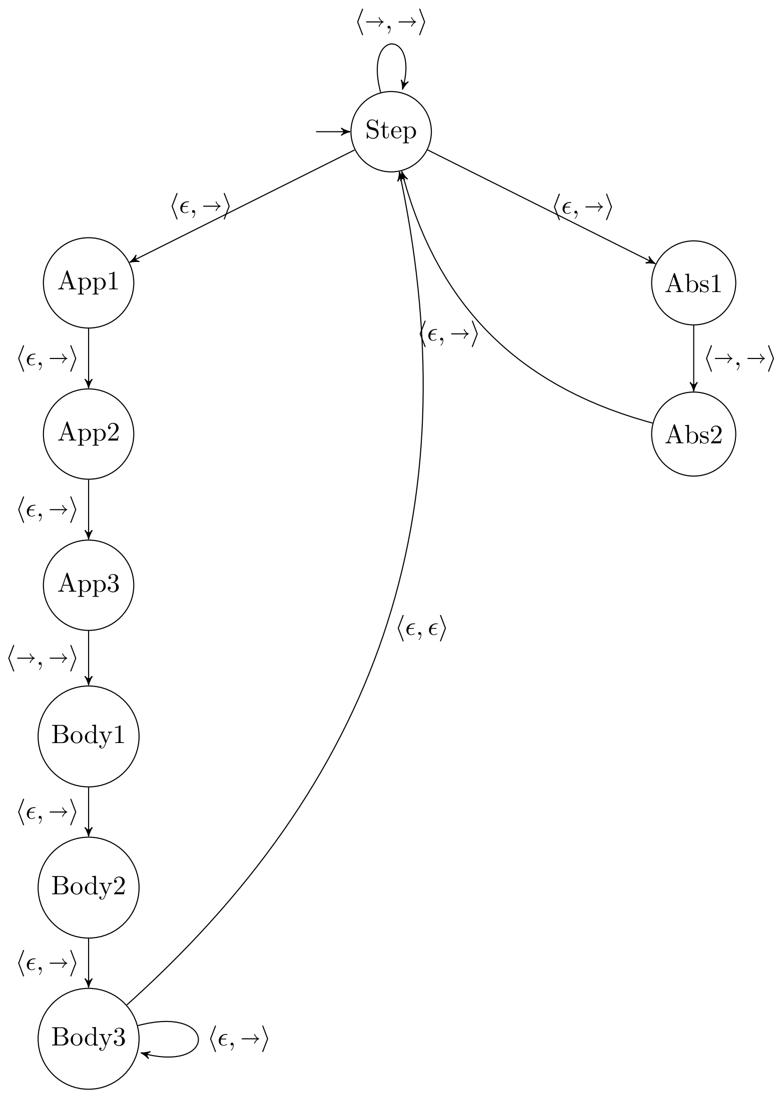


Figure 4.8: Stage automaton for closure conversion

we need to make a step between stages g_1 and g_2, with edge $\langle \epsilon, \rightarrow \rangle$, it will have us going from having $s_1 \approx^{g_1} s_2$, and $s_2 \rightarrow s_2' \; ! \; i$, to produce $s_1 \approx^{g_2} s_2'$.

It should be noticed in the code, that we are adding several new variables, such as f_0, f_e, and a. These will need to be made fresh within the program. Our stage can include which variables these are, so that we can easily express code relations, but these working variables will also need to be accounted for in the relation. Much as we had an environment of globals ρ_g, we will want to assign to each such variable a particular value. Then we should have that that value is in the environment, is the expected shape, and, in the heap, has the expected store value. That is, for instance, in App1, we will have just bound the f_0 component. Thus, the stage will have the variable f_0, and a corresponding value v_{f_0}, which should be an address pointing to an appropriate (empty) **sclos** store value.

Our address relation will be somewhat more complex. Now, one address in the source (a closure), may be mapped to several addresses in the target (the explicit closure of both a lambda and an environment record). We will expect that not only does this agree with the domains of the heaps, but that it lines up with the expected store values. Likewise, we will maintain the property that the image of the (address-set) closure corresponds with the closure of the image. In this case, the image may send one address of a closure to the three corresponding addresses of an explicit closure. The roots of the state will not contain a partial portion of the closure, except when performing the application itself. We can recognize, we do not need to generalize over address-sets as before, in order to be general enough for the proof. It may still be useful to do so, as it avoids dirtying the stepping proofs when the heap structure does not change. During the construction of a closure, we may have additional addresses in the heap which are not in the address relation. These may need to be handled separately when assembling the final space-safety proof.

We have two sorts of situations to consider for space-safety: the construction and the application. If we line up the source step with the construction of the tuple, then we can get by with a closer relation. The variable f_0 costs 2 units of space, one for the base object cost, and one for the variable. The tuple f_e costs as much as the original closure. Adding the tuple f creates an overhead of one unit for the tuple header (since we've already accounted for its components), and one for the address. Since at that point we can include it into the address relation, the size of those three, relative to the ideal closure, will be the multiplicative factor. From the above, we have a cost of $2 + 2 + n + 1 = 5 + n$ in the target, where the source has size $1 + n$, for the closure and pointer. In the edge case of $n = 0$, this creates a multiplicative overhead of 5, which may seem quite large, but this only indicates that our source semantics were optimistic. It is perfectly valid for the efficiency of the transformation, and could be improved if desired, since carefully including the original pointers in both would reduce it to $\frac{6+n}{2+n}$, reducing the constant to 3; we could keep a more detailed relation between address-sets to do so. Optimizing the transformation for empty closures (which need less change) would likewise improve it. As for applications, there is a finite overhead due to the argument tuple along with the extra variables. At maximum, the argument tuple a has size 3. While projecting components from its environment f_e, there is overhead to keep the argument e around. We could account for this as a constant additive factor, or, noticing that eventually all free variables will appear in the environment and thus have costs as variables, we may use a two-times multiplier on the cost of code. In any case, with a multiplicative constant at most 5, and a fixed-per-program additive constant, the closure conversion will be space efficient.

As far as preservation of the relation, it is a good bit simpler than with globalization. Unlike globalization, all higher-order data must be in the same stage (Body1 above). All code in closures is related as in Body1, where the next step is to unpack the environment variable from the argument tuple. As every variable is now computed from the closure, there are no free variables, and the environment of such a closure thus needs nothing in it, and has no cost. For other data, the definitions should be standard, outside of construction stages, each corresponding variable maps to a corresponding address, where the root of the three-address closure object is considered.

Chapter 5

Related Work

5.1 Proper Tail Recursion and Space Efficiency

In 1998, Clinger[3] presented a formal presentation of the relatively informal definition of "proper tail recursion", in addition to contrasting it to several other models. Working by way of a direct interpreter for a Scheme subset, several space semantics could be presented by counting reachable allocations. With this in hand, he was able to present an ordering of several space semantics, following from an Algol-like stack through to the safe-for-space condition of Appel (that is, containing both tail-call elimination, and collection of conditionally used variables.

Significantly, Clinger also considered the usage of linked closures instead of flat closures. While Appel's condition would in general not work for linked closures, in some cases, they may use asympotically less space than even the safe-for-space condition with flat closures. This asymptotic improvement appears because the closures with a linked representation may share closure size. This improvement only shows up when considering asymptotic improvements over the class of all programs; allowing the multiplier to vary from one program to the next causes the effect to disappear (as the degree of sharing is fixed per program).

5.2 Space Profiling Semantics of the Call by Value Lambda Calculus and the CPS Transformation

In 1999, Minamide provided[17] a simple space semantics for the call-by-value lambda calculus, and proved a very strong notion of efficiency for the CPS transformation (that is, the translation of a direct-style program to continuation passing style representation) [17]. This proof was not formalized, but was one of the first (if not the first) published proofs of space safety for a global transformation. Regardless, this transformation did not drastically permute the heap, and likewise was considered only for terminating programs.

His notion of efficiency is slightly stronger than the notion in this work, by requiring the additive term to be constant, regardless of program. A further work expanded the definitions to include the notion of weakly space efficient[18], which is incomparable with our notion of space-efficiency; they include a program's code size, and define a weakly space-efficient transformation to be one whose space increase is bounded by a polynomial in the program size. Then, a particular example was shown not to meet this standard, namely one where the type information was present at runtime, but was not counted in the size of the source program. This example failed because the size of types may be exponential in the size of the program, which could only be fixed by including the size of the types in the total size of the program.

5.3 Efficient and Safe-for-Space closure conversion

n 2000, Shao and Appel presented a method[24] for improving the (practical) efficiency of closure conversion, compared to flat closures. This method has additional sharing of common elements compared to flat closures, improving both space and time efficiency. Their work did not include a formal proof of its own correctness, but did include empirical data from various tests. Nevertheless it was a relatively early paper handling potential issues with space-safety, with an algorithm for better representation.

5.4 Closure Conversion is Safe-for-Space

Paraskevopoulou and Appel[19] presented a fully verified safe-for-space flat closure conversion for use in the CertiCoq compiler. This presentation has several differences from this work: time safety is included, in addition to space safety; The space safety of programs is proven only for an entire program run (completed or not) with no I/O or other checkpoints present; garbage collection is performed only at function entry, not at every program point, as will be the case in this work. In comparison, the environment layout, and the space semantics are the same.

The choice to perform garbage collection only at specified points is a helpful simplification compared to nondeterministic collection, and agrees fully on space-efficiency when we allow an additive constant to vary by program (since only a finite amount of data may be allocated before collection in their language).

Mechanically, their method proceeds by way of logical relations, in contrast to the more syntactic relations in this proof. In handling garbage collection, their relations universally quantify over all possible heaps, so long as those heaps agree on the roots of the environment (at function entry). Furthermore, their machines are realistic for the implementation strategy, whereas the machines in this work accounted for space live only when computing the space, rather than actually performing any garbage collection. There are several other distinctions: due to the usage of logical relations, their relations are indexed by a maximum run-time as well as a maximum lookup depth. Instead, the definitions in this work use the full closure, exploiting the finiteness of heaps. Since the relations in this work are syntactic, there is no need to have them indexed by a maximum step.

Chapter 6

Conclusion

Our final development, based on justifications of states met all of the goals expected. It correctly handles both programs with I/O, as well as non-terminating programs, and enforces a reasonable degree of temporal locality in the justifications. Because our proof used this method, we know that every potential sequence of steps in the program is justified only by the same sequence, and that every expense must be justified by an original expense that is not too far away.

This work succesfully proved the space-safety of lifting small values to a global scope, as well as the space-efficiency of doing so when the globals were correctly not counted in the size of closures, identifying a flaw with using the common definitions of using closures even at the top-level scope. Our proof uses the full justification map, ensuring that it meets the locality properties we expect. In order to complete this proof, we had to develop a direct relation between addresses in the source and in the target, including required aliasing details, and because of our quantification over different sets, we were able to easily keep the relation established as the exact relation between sizes fluctuated. Our space-safety relation was only required to expand upon passing over object allocations, never merely a garbage collection point. This allows us to handle more problems than could be handled by a relation which only counted the true live data, without requiring us to repeatedly update it as the proof progressed.

6.1 Future Work

While this work addressed a particular case of space safety, as shown earlier with Clinger's ordering and through the various forms of garbage collection, there are many more semantics. Expanding definitions to handle space-efficiency for these other conditions, such as a region-based system, stacks, or the access conditions should be a next step. Even languages which do not meet the exact standards of space safety deserve the ability to prove transformations to be efficient. We may also be open to the idea that tighter space semantics are desirable. In accounting for common space leaks, it may be beneficial to move towards the condition of access, or even the simple tuple-projection optimizations present in GHC[27], where a small amount of analysis can prevent common space leaks.

Although we may have improved the ability to reason about the performance of code, it should be noted that improving space is not enough for all scenarios. We should also aim for proving some performance-isomorphisms. The reason may be elucidated by the following: Suppose we have a compiled, target program P, being compiled from a more abstract source program $\hat{P}$. A programmer who knows the source program semantics may reason that they can make an improvement from $\hat{P}$ to some equivalent but more efficient program $\hat{P}'$. The space usage might still actually worsen, since there is no a-priori mechanism which ensures that when we have $\hat{P} \geq \hat{P}'$, we will necessarily have $P \geq P'$. Doing so corresponds to filling in the bottom edge of the below diagram.

$$\begin{tikzcd}
\hat{P} \arrow[r] \arrow[d, "f_P"'] & \hat{P}' \arrow[d, "f'_P"'] \\
P \arrow[r, dashed, dash, "?"] & P'
\end{tikzcd}$$

We can only rarely expect our transformations to always preserve this, since they must be dependent on syntactic properties. One particular case in which it does work is if the transformation is actually an space-equivalence on the two programs: that is, every source state is also justified by a target state (allowing different constants). Then we may get the corresponding direction by composition. It should be the case that some concretion passes, such as closure conversion, and changes in garbage collection time will be isomorphisms, but this must be shown. For such isomorphisms, both formal and informal proofs may treat the two representations as genuinely equivalent.

This work has not addressed running time of programs. There has been some work on this, as it was included in the work by Paraskevopoulou and Appel[19]. Regardless, it should fit into the framework presented here: although space usage examined only the destination state, time, or other costs can consider the entire time spent on steps since the last event. Similarly, requiring relations to hold between the number of steps or the time of the justified step and the justifying step may also work to better model computation-heavy (as opposed to I/O-heavy) programs. The definitions for space-efficiency/safety have made use of only a small portion of the data associated with reachable states. In particular every such state has a unique predecessor (or is the start), and as such has a finite length which can be compared similarly to time costs.

We should re-examine some type systems or other languages which have restricted space or time complexity. The Parsimonious Lambda Calculus [15,16] has as well-typed terms only polynomial time, logarithmic space computations. A variation of Lafont's Soft Linear Logic with a special conditional similarly characterizes polynomial space programs [6]. The basis for some of these systems, in particular those based on Soft/Light/Elementary linear constructions, is on the size of proof terms, and the complexity of cut elimination. We know that the size of proof terms, fully expanded, is not exactly the same as the size of terms, when sharing is involved. A simple example may be seen in natural numbers. For Church numerals, which have little non-trivial sharing, the predecessor operation on n takes time and space proportional to $O(n)$, but for inductive datastructures, such an operation requires constant time and constant space.

The recent paper “Call-by-Need Is Clairvoyant Call-by-Value” by Hackett and Hutton[8] represented call-by-need in terms of choices made on top of a call-by-value base. I would suspect that the situation for call-by-need terms versus call-by-name or call-by-value terms should resemble the situation for shared space usage, and the time costs could be modeled under a similar aliasing system.

This language, although it had a few features, was yet unrealistic. A modern programming language, even including the venerable Standard ML, will include several features which were not present in this machine. Mutable references were not present in this language, but are a feature of many programming languages. In general, it may increase the complexity of proofs notably under this methodology, as the quantification over address sets now needs to account for updates to the record (which may change the closure). Perhaps this is apt though, as the presence of mutation changes the available transformations that we can make safely, as was seen with MLton’s flattening passes.

Bibliography

[1] Andrew W. Appel. 1992. *Compiling with continuations*. Cambridge University Press, New York, NY, USA.

[2] David R. Chase. 1988. Safety considerations for storage allocation optimizations. In *PLDI*.

[3] William D. Clinger. 1998. Proper tail recursion and space efficiency. *SIGPLAN Not.* 33, 5 (May 1998), 174–185. DOI:`https://doi.org/10.1145/277652.277719`

[4] Yannick Forster, Fabian Kunze, and Marc Roth. 2019. The weak call-by-value lambda-calculus is reasonable for both time and space. Retrieved from `http://arxiv.org/abs/1902.07515`

[5] Pascal Fradet. 1994. Collecting more garbage. *SIGPLAN Lisp Pointers* VII, 3 (July 1994), 24–33. DOI:`https://doi.org/10.1145/182590.182417`

[6] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. 2012. An implicit characterization of pspace. *ACM Trans. Comput. Logic* 13, 2 (April 2012), 18:1–18:36. DOI:`https://doi.org/10.1145/2159531.2159540`

[7] Benjamin Goldberg and Michael Gloger. 1992. Polymorphic type reconstruction for garbage collection without tags. *SIGPLAN Lisp Pointers* V, 1 (January 1992), 53–65. DOI:`https://doi.org/10.1145/141478.141504`

[8] Jennifer Hackett and Graham Hutton. 2019. Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.* 3, ICFP (July 2019), 114:1–114:23. DOI:`https://doi.org/10.1145/3341718`

[9] Niels Hallenberg, Martin Elsman, and Mads Tofte. 2002. Combining region

inference and garbage collection. *ACM SIGPLAN Notices* 37, (May 2002). DOI:`https://doi.org/10.1145/512529.512547`

[10] Haruo Hosoya and Akinori Yonezawa. 1998. Garbage collection via dynamic type inference - a formal treatment. In *In proc. Second international workshop on types in compilation (tic'98), volume 1473 of lect. Notes in computer sci*, 215–239.

[11] INRIA. 2019. The Coq proof assistant. Retrieved December 28, 2019 from `coq.inria.fr`

[12] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A safe dialect of c. In *Proceedings of the general track of the annual conference on usenix annual technical conference* (ATEC '02), 275–288. Retrieved from `http://dl.acm.org/citation.cfm?id=647057.713871`

[13] Well-Typed LLP. 2016. Sharing, space leaks, and conduit and friends. Retrieved February 25, 2019 from `https://www.well-typed.com/blog/2016/09/sharing-conduit/`

[14] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. 2017. Compiling without continuations. *SIGPLAN Not.* 52, 6 (June 2017), 482–494. DOI:`https://doi.org/10.1145/3140587.3062380`

[15] Damiano Mazza. 2015. Simple parsimonious types and logarithmic space. In *CSL*.

[16] Damiano Mazza and Kazushige Terui. 2015. Parsimonious types and non-uniform computation. In *Automata, languages, and programming*, 350–361.

[17] Yasuhiko Minamide. 1999. Space-profiling semantics of the call-by-value lambda calculus and the cps transformation. *Electr. Notes Theor. Comput. Sci.* 26, (December 1999), 105–120. DOI:`https://doi.org/10.1016/S1571-0661(05)80286-5`

[18] Yasuhiko Minamide and PRESTO. 2001. A new criterion for safe program transformations. *Electronic Notes in Theoretical Computer Science* 41, 3 (2001), 20–34.

DOI:https://doi.org/https://doi.org/10.1016/S1571-0661(04)80871-5

[19] Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure conversion is safe for space. *Proc. ACM Program. Lang.* 3, ICFP (July 2019), 83:1–83:29. DOI:https://doi.org/10.1145/3341687

[20] Simon L. Peyton Jones. 1987. *The implementation of functional programming languages (prentice-hall international series in computer science).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[21] Iris Project. 2019. Coq-std++. Retrieved December 2, 2019 from https://gitlab.mpi-sws.org/iris/stdpp

[22] Iris Project. 2019. Iris project. Retrieved December 2, 2019 from https://iris-project.org/

[23] Raphaël L. Proust. 2017. *ASAP: As Static As Possible memory management.* University of Cambridge, Computer Laboratory. Retrieved from https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-908.pdf

[24] Zhong Shao and Andrew W. Appel. 2000. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.* 22, 1 (January 2000), 129–161. DOI:https://doi.org/10.1145/345099.345125

[25] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. 2010. Space profiling for parallel functional programs. *Journal of Functional Programming* 20, 5-6 (2010), 417–461. DOI:https://doi.org/10.1017/S0956796810000146

[26] Mads Tofte, Stack Profile, and Stat Option. 1997. Programming with regions in the ml kit.

[27] Philip Wadler. 1987. Fixing some space leaks with a garbage collector. *Softw., Pract. Exper.* 17, (1987), 595–608.

Appendix A: Standard ML

Code samples in this work are presented in Standard ML.

Standard ML is a garbage-collected, call-by-value, impure, statically typed, functional programming language. That means that objects are automatically released when unneeded, expressions are evaluated immediately, and mutations, I/O and other effects may occur immediately with evaluation of a term. Unlike many languages, well-typed Standard ML programs do not need type signatures, so we will often omit type signatures. It is notable for having a formal definition of its semantics. There are numerous compilers and interpreters for Standard ML. As a historical note, much of the existing work regarding space safety has targeting compiling either Standard ML or Scheme. These languages are both simple, higher-order call-by-value functional languages, with a wealth of compilers and compiler research. In some cases, certain features are incompatible with the strictest definitions for space-safety, or require additional effort to compile correctly. A brief overview of relevant syntax will be presented below.

Comments are delimited by the glyphs `(*` and `*)`. We will use these to make annotations on code.

Variables (and other patterns) may be declared using both of the following syntaxes: `val` defines a simple expression, which will be evaluated immediately. The result is bound to the variable x. Re-assignments are treated as fresh bindings, not mutation.

```
val x = e
```

Using `fun` defines a function. One or more arguments or patterns may appear to the right of the variable, which then become arguments. The function name is available in the body for recursive calls.

```
fun f x = x + 1
fun g (x, y) = x * y
```

The group `(x, y)` above denotes a tuple pattern, so `g` actually takes one argument, which is itself a tuple object containing two fields. When `g` is evaluated, those fields are deconstructed and the values bound to the variables `x` and `y`. Tuple expressions are written identically to patterns, and `(e1, e2)` denotes an expression where e1 is evaluated, then e2 is evaluated, then their results are stored in a tuple, and that value is returned.

Applying functions is simple, we simply juxtapose the function and its arguments (with spacing if necessary). Thus the following are all valid function calls:

```
f 4
f(4)
g tup (* if tup is a variable in scope which is an appropriately typed *)
g (3, 4)
```

The expression syntax `let ... in ... end`, includes any number of declarations in the first section between `let` and `in`, then contains another expression between `in` and `end`. Each declaration in the first section is evaluated in turn, extending the environment, then the last expression is evaluated in the environment, and returned. For example, see the following

```
let
  val i = 5
  val j = i * 2
in
```

```
  i * j
end
```

This expression will calculate the value 50, by first binding the value of the expression 5 to `i`, then evaluating the expression `i * 2`, and binding it to `j`, then evaluating `i * j`, and returning that as the value of the `let` expression.

Lists are constructed using either the constructors `[]` (nil), and `::` cons, or with list syntax, placing several expressions in between square brackets `[e1, e2, ..., en]`.

Lists are a datatype, and can be deconstructed using a case statement, `case ... of p1 => e1 | p2 => e2 | ... pn => en`. Each pattern in sequence is tested, until one can be successfully applied, then that branch is taken. They can be seen as a generalization of an if statement, where `if e1 then e2 else e3` is de-sugared to `case e1 of true => e2 | false => e3`. Patterns can be nested, and a wildcard symbol _ can be used for a position that does not need binding. For example, combining some of these features, we may have

```
fun len l =
  case l of
    _ :: xs => 1 + len xs
  | [] => 0
```

These patterns are used identically to the function arguments and bindings. We could equally well have written `g` above as:

```
fun g t =
  case t of
    (x, y) => x * y
```

Likewise, function definitions may also perform pattern-matching directly, via additional lines with a pipe and the function name. For instance, the function `len` above may be rewritten to the shorter form:

```
fun len [] = 0
  | len (_ :: xs) = 1 + len xs
```

Standard ML, as a functional language, has first-class functions. Functions may take arguments, and return values which are callable functions. Moreover, there is a literal function (lambda) syntax which we may use occasionally: `fn p => e`. Going even farther with `g` above, we may even reduce it to either of the following:

```
val g =
  fn t =>
    case t of
      (x, y) => x * y
val g =
  fn (x, y) => x * y
```

A few portions of the work may mention references. In Standard ML, references are mutable cells which can be created to hold any particular type of value with the `ref` function, and can be assigned with `:=`, and read with `!`, as follows. Also, it is useful in imperative code to sequence computations (with effects), while discarding their results; this function will be performed by a semicolon `;`.

```
fun foreach (xs, f) =
  case xs of
    [] => ()
    (x :: xs) => f x; foreach (xs, f)

fun len2 xs =
  let
    val i = ref 0
    val () = foreach (xs, fn _ => i := !i + 1)
  in
    !i
  end
```

Appendix B: Problematic transformation with grouping-definition

This appendix contains an example pair of programs which are safe-for-space under the final definition, and which are hopefully "intuitively" safe, but which cannot be proven to be so with the grouping condition, at least by any simple proof. I suspect that there are simpler loop fusions which exhibit this behavior, and that the computable constraint is unnecessary, but this should be sufficient to show potential issues with the condition.

The condition of the proof being computable is somewhat informal, but contains all axiom-free proofs in Coq. For such a proof, all decisions must be computed from their input, so in particular, the branch of every disjunction $P \vee Q$, and the witness of every existential $\exists x, Px$ may computed by an algorithm of the input. This ideally should not be a necessary condition, but is used to ensure that the proof does not depend on the genuine space usage of the programs, and given that the main proofs of this work, and the proof below are computable, we ought to expect this to suffice.

Claim 1. *There exist space-preserving transformations with no computable proof of space-safety under the finite grouping condition (the $\gtrsim_f$ relation on traces)*

We can show that there is no general proof of soundness for a transformation by providing a particular program which will not have such a proof.

We will imagine a hypothetical transformation that moves some code out of a loop, and offsets the iterations for portions of a long-running loop with no side-effects (besides possible non-termination). The example below will illustrate as we are only concerned with one instantiation. Note that this is sound regardless of what the code is (since the effect of non-termination cannot be distinguished from different instances; i.e. it is a commutative effect). Likewise, the space usage of said code will be completely unchanged as long as all iterations are performed.

Suppose that there were a computable proof of space-safety of the general transformation.

Then by the definition of complexity/space-safety there exists constants a and b, such that for each grouping $s_1, \ldots s_m$ and $t_1, \ldots, t_n$, we have $a \cdot max_i(|s_i|) + b \geq max_j(|t_j)$.

Since the proof is computable, and since space usage is a semantic property of programs, by Rice's theorem there are programs whose space usage is not known to be bounded by any constant for any input (to any estimation present in the proof). Consider two such programs f and g (from natural numbers to natural numbers; their space usage is discarded after evaluation). Thus, the proof must treat each such space usage as opaque (although technically individual states in execution can be inspected, there will be some 'core' whose space usage is still opaque). A loop condition should also be chosen so that there is some chance of non-termination, though it suffices to use user input to ensure both possibilities can occur in the same program (and different traces), so there is no need for a clever choice. We will also require that the loop condition, here rand, should be unknown to the proof; otherwise it can again group the entire finite length into one block. Although in any case, it may certainly not be too desirable.

Now imagine the transformation changes the code:

```
let
  val n = ... (* read natural number from user input *)
  val i = ref 0
in
  while (!i < 2 or rand(!i) > 0 or n < 0)
    (f(i);
     g(i);
     i := !i + 1)
end
```

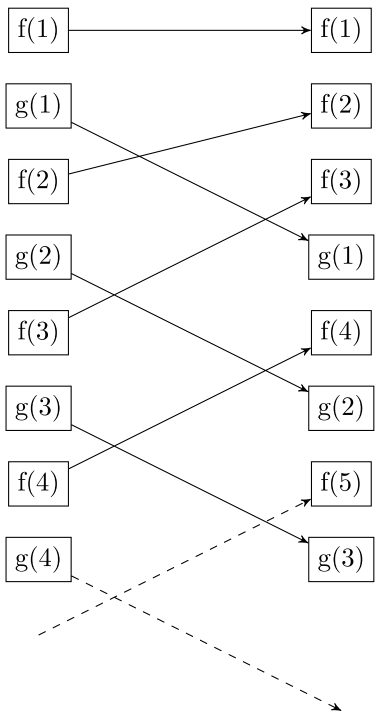
into the code:

```
let
  val n = ... (* read natural number from user input *)
  val i = ref 0
in
  f(0);
  f(1);
  while (rand(!i) or n < 0)
    (f(i + 2);
     g(i);
     i := !i + 1);
  g(n);
  g(n+1);
  ()
end
```

By construction, the space usage of each $f(i)$ and $g(i)$ must be opaque computable functions $s_f(i)$ or $s_g(i)$ to the proof. Informally, since any of these terms may be more than $a \cdot s_{f(j)} + b$ or $a \cdot s_{g(j)}$ for other j, they must always be justified by the corresponding constant[1]. Then, we have the following correspondance, where whenever a state is in the

[1]More specifically, we can construct computable functions such that this condition does not hold at the particular finite points desired, again adversarially to the proof, to show that any other grouping is invalid for a general computable function.

target group, the corresponding state must appear in the source group. We also have the reverse: each source state must appear in the target, since otherwise there might never again be a source state which justifies the target. Although we could get through a few steps, we will eventually have a source state which was already consumed, and thus cannot be justified. But, in the case of an infinite loop, this grouping can never be finite, since whenever it has $g(i)$ in the source portion, it also has $g(i+1)$ through the application of the above two constraints plus the requirement on ordering. Below are the required pairings for an infinite loop (n < 0).



Since there can be no grouping subject to the constraints for this program, there must not be any space-safety proof using finite grouping for the pair of programs above.