Shrink Fast Correctly!

Olivier {Savary Bélanger}
Princeton University
olivierb@princeton.edu

Andrew W. Appel
Princeton University
appel@princeton.edu

ABSTRACT

Function inlining, case-folding, projection-folding, and dead-variable elimination are important code transformations in virtually every functional-language compiler. When one of these reductions strictly reduces the size of the program (e.g., when the inlined function has only one applied occurrence), we call it a *shrink reduction*. Appel and Jim [1] introduced an algorithm to perform all shrink reductions (producing a *shrink normal form*) in quasilinear time. They proved confluence but not correctness.

We have implemented this algorithm as part of an end-to-end verified compiler for Gallina, the specification language of the Coq theorem prover. We have given the first proofs of these properties: correctness with respect to contextual equivalence, reduction (in one pass) of all administrative redexes with one applied occurrence introduced by CPS conversion, and termination. The correctness and termination proofs are machine-checked in Coq.

Because we use a pure functional language without imperative array update, our implementation is $O(N \log N)$ rather than $O(N)$. Still, it's quite fast: we give performance results on some nontrivial benchmarks.

ACM Reference format:
Olivier {Savary Bélanger} and Andrew W. Appel. 2017. Shrink Fast Correctly!. In *Proceedings of International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, 9 – 12 October 2017 (PPDP'17),* 12 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

If the program has bugs, why bother proving the compiler correct? If the compiler has bugs, why bother proving the program correct? Verified source programs deserve verified compilers, and vice versa. We [2] are building CertiCoq, a verified-correct compiler for Coq—that is, for the functional language that's part of Coq's Gallina specification language. All compiler phases will be proved to preserve observable behavior from each intermediate language to the next, with machine-checked proofs in Coq. The user can prove a program correct in Coq, then the verification of CertiCoq guarantees that this program compiled to machine-language has the behavior that the user verified at the source level.

We want this compiler to be not only correct but *efficient*. We have improved, implemented, and proved correct an algorithm [1] that in one fast pass performs dead-variable elimination, inlining of single-use functions, projection folding, and case folding. To do this in one pass, the algorithm must efficiently and incrementally update its dataflow information and usage counts. This additional complexity is made worthwhile by the cascading reduction opportunities which appear as other reductions are performed. We are not the first to implement this "shrink-reduction" algorithm, but we are the first to prove it, or an implementation, correct.

In a functional language with immutable data structures—such as Gallina, ML, Haskell—several optimizations are particularly important: **function inlining** (β-reduction); **case-folding**, compile-time evaluation of case statements when the discriminant value can be statically determined; **projection-folding**, compile-time fetching of projections of tuple-fields (or generally, fields of inductive data constructors) when the tuple can be statically determined; and **dead variable elimination**. The reason these are more important in functional languages than in traditional imperative languages is that there are far more opportunities: functional languages and their compilers use functions more heavily, and folding of field-projections is possible only when the record-fields cannot have been updated with new values. Also, many compiler transformations introduce β-redexes. For example, simple CPS transformations introduce many so-called *administrative (β-)redexes* which can be safely reduced to recover a more compact program.

It is important to do all these optimizations *together,* because one may produce new opportunities to do another.

For example, in

$$\text{let } f\ x := (\text{match } x \text{ with } O \Rightarrow M\ ;\ S\ \_ \Rightarrow N) \text{ in } f\ O$$

we can inline f resulting in

$$\text{match } O \text{ with } O \Rightarrow M\ ;\ S\ \_ \Rightarrow N$$

at which point the constructor O is exposed and the case-construct can be folded down to M. Then, since the expression N has disappeared, some of the free variables of N (bound in some context external to the entire *let* expression) may now be dead, permitting dead-variable elimination.

Case-folding, projection-folding, and dead-variable elimination are always worth doing, because they make the program smaller and faster. Function inlining usually makes the program faster, but if there are many uses of the function, it may make the program bigger. Inlining a function that has only one applied occurrence will make the program smaller and faster, because the function-definition is now dead and can be deleted. Appel and Jim [1] described this class of optimizations (case-folding, projection-folding, dead-variable elimination, and inlining functions with one applied occurrence) as *shrink reductions.*

A traditional function-inliner or dead-variable optimizer makes one static-analysis pass over the program, counting applied occurrences and learning which functions are worth inlining; then

PPDP'17, 9 – 12 October 2017, Namur, Belgium

another pass performs the transformations; then (because the transformations may have enabled new optimizations) repeats the analysis pass, then another optimization pass, and so on until the analysis pass yields no new optimizations to perform.

Appel and Jim described an efficient algorithm that performs (almost) all the possible shrink reductions, even cascading ones, in one linear-time pass. This is a *quasilinear time* algorithm: in linear time, it typically reduces to shrink-normal form, but sometimes leaves a very small number of shrink redexes to be reduced in a second pass, or very rarely in a third pass. Appel and Jim also described a fully linear-time algorithm that heavily uses imperative graph-update, which is less convenient to implement in a functional programming language. Kennedy [3] improved, implemented, and measured the fully linear-time algorithm, and reported excellent performance.

No one has done a formal proof (machine-checked or otherwise) of correctness of either of these algorithms, or of any efficient algorithm for approaching shrink-normal form. In this paper, we prove the correctness of an implementation (in Coq) of Appel and Jim's quasilinear-time algorithm, and demonstrate that it achieves excellent performance.

Instead of presenting a monolithic proof of correctness, we modularize it into three layers: We first prove that the program `contract_top` performs reductions according to a system of shrink-rewrite rules ($\rightarrow_C^*$). We then prove this system to be a specialization of a second, more general rewrite system ($\rightsquigarrow_C^*$), describing a wider class of optimizations. Finally, we prove that the general rewrite system only relates equivalent programs($\cong^{\exp}$).

$$\cong^{\exp} \overset{Thm.3.1}{\Longleftarrow} \rightsquigarrow_C^* \overset{Thm.4.1}{\Longleftarrow} \rightarrow_C^* \overset{Thm.5.3}{\Longleftarrow} \text{contract\_top}$$

The paper is organized as follows.

§2 We present our object language, showing its syntax, its semantics and our notion of equivalence on environment, terms and values.

§3 We present a series of rewrite rules admissible by general reduction. We use these rules to form a rewriting system, which we prove correct according to our notion of equivalence.

§4 We refine the rules to admit only rules that shrink the overall size of the program. We prove that the rewriting system formed by the refined rule (so called shrink rewrites) is included in the general rewriting system.

§5 We present an algorithm that reduces most shrink redexes in quasilinear time, evaluate its performance and prove that it only modifies the term according to the shrink-rewrite system, which, composed with the previous proofs, shows that the algorithm is correct.

§6 We show benchmark measurements on realistic, substantial programs that demonstrate that the shrink-reducer is really fast, and very effective.

Contributions:

- We improve on the shrink inlining algorithm presented in Appel and Jim [1], simplifying the presentation and preserving a more accurate count of variable occurrences.
- We prove the algorithm correct in Coq with respect to contextual equivalence, which Appel and Jim did not do even on paper.
- We prove correctness not just of an algorithm, but of an *implementation* of the algorithm, an efficient working functional program.
- We show that the algorithm reduces all administrative redexes [4] with one applied occurence introduced by the CPS transformation, in a single pass. This is important because a simpler CPS transformation is significantly easier to implement and prove correct than one that reduces administrative redexes [5].
- We have implemented this as part of CertiCoq, and our proof composes with the rest of the compiler to provide an end-to-end correctness guarantee.

The Coq development with our implementation and proofs is available at:
https://www.cs.princeton.edu/~appel/shrink-fast-correctly.tar

2 BACKGROUND

Syntax

$$\begin{array}{llll}
\text{(Function Def'n)} & fd & ::= & f\ (\vec{x})\ =\ e \\
\text{(Branch)} & b & ::= & c \Rightarrow e \\
\text{(Expression)} & e & ::= & \ \text{let } x = \text{Con } c\ \vec{y} \text{ in } e \\
 & & & \mid \text{let } x = \text{Prim } p\ \vec{y} \text{ in } e \\
 & & & \mid \text{let } x = \text{Proj}_n\ y \text{ in } e \\
 & & & \mid \text{App } x\ \vec{y} \\
 & & & \mid \text{let } \vec{fd} \text{ in } e \\
 & & & \mid \text{match } x \text{ with } \vec{b} \\
 & & & \mid \text{halt } x \\
\text{(Value)} & v & ::= & (c,\ \vec{v}) \\
 & & & \mid (\rho, \vec{fd}, x) \\
\text{(Environment)} & \rho & ::= & \cdot \\
 & & & \mid \rho, x \mapsto v
\end{array}$$

Figure 1: Syntax of the Object Language

Our object language is a continuation-passing-style functional language with mutually recursive functions and pattern-matching. Figure 1 shows its syntax. "let x = Con $c\ \vec{y}$ in e" binds the constructor c applied to arguments $\vec{y}$ to variable x in expression e. "let x = Prim $p\ \vec{y}$ in e" binds the result of the primitive operator p on arguments $\vec{y}$ to variable x in expression e. "let x = $\text{Proj}_n\ y$ in e" binds the nth projection of y to variable x in in expression e. "App $x\ \vec{y}$" applies function x to arguments $\vec{y}$. "match x with $\vec{b}$" matches the constructor c of x with the right branch $(c \Rightarrow e) \in \vec{b}$.

To simplify the presentation, we assume that branch patterns do not overlap and that function names within each bundle are distinct. "halt x" terminates computation by returning the value bound to x.

The semantics of our object language is given through a big-step, environment-based judgment $\rho \vdash e \Downarrow_k v$ evaluating expressions e in environment ρ into value v in at most k β-reductions. We will sometimes omit the argument k and just write $\rho \vdash e \Downarrow v$ when the cost is inconsequential. The environment maps variables to values. A value is either a constructor c with its arguments $\vec{v}$ or closure including a function's body e with its parameters $\vec{x}$ and an environment ρ providing values for the function's free variables. Figure 2 shows the evaluation rules.

$$\frac{\rho(x) = c\ \vec{w} \quad (c \Rightarrow e) \in \vec{b} \quad \rho \vdash e \Downarrow_k v}{\rho \vdash \mathsf{match}\ x\ \mathsf{with}\ \vec{b} \Downarrow_k v}\ \textsc{e\_match} \qquad \frac{\rho(y) = c\ \vec{w} \quad \rho; x \mapsto w_n \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e \Downarrow_k v}\ \textsc{e\_proj}$$

$$\frac{\rho(f) = (\rho',\ \vec{fd},\ f) \quad (f\ (\vec{x})\ =\ e) \in \vec{fd} \quad \forall_{y_i \in \vec{y}}, \rho\ y_i = v_i \quad \rho'; f_i \mapsto (\rho', \vec{fd}, f_i); x_i \mapsto v_i \vdash e \Downarrow_k v}{\rho \vdash \mathsf{App}\ f\ \vec{y} \Downarrow_{k+1} v}\ \textsc{e\_app}$$

$$\frac{\forall_{y_i \in \vec{y}}, \rho(y_i) = w_i \quad \rho; x \mapsto (c, \vec{w}) \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ e \Downarrow_k v}\ \textsc{e\_constr} \qquad \frac{\forall_{y_i \in \vec{y}}, \rho(y_i) = w_i \quad f\ \vec{w} = w \quad \rho; x \mapsto w \vdash e \Downarrow_k v}{\rho \vdash \mathsf{let}\ x = \mathsf{Prim}\ f\ \vec{y}\ \mathsf{in}\ e \Downarrow_k v}\ \textsc{e\_prim}$$

$$\frac{\rho; f_1 \mapsto (\rho, \vec{fd}, f_1); \ldots; f_n \mapsto (\rho, \vec{fd}, f_n) \vdash e \Downarrow_k v \quad \text{where } \mathsf{names}(\vec{fd}) = \{f_1, \ldots, f_n\}}{\rho \vdash \mathsf{let}\ \vec{fd}\ \mathsf{in}\ e \Downarrow_k v}\ \textsc{e\_fun} \qquad \frac{\rho(x) = v}{\rho \vdash \mathsf{halt}\ x \Downarrow_k v}\ \textsc{e\_halt}$$

Figure 2: Evaluation rules of the object language

In our object language, pattern-matching is broken into two operations. First, our case constructor "match x with $(c_1 \Rightarrow e_1, \ldots, c_n \Rightarrow e_n)$" determines which pattern c_i the construction bound to variable x matches, and proceeds to evaluate e_i, as seen in E_MATCH. Then, projection constructs "let x_1 = $\mathsf{Proj}_1\ x$ in...let x_m = $\mathsf{Proj}_m\ x$ in" are used to bind variables to the m arguments of c_i which will be replaced by the right values when evaluated as shown in rule E_PROJ.

ML's and Haskell's syntax and type systems connect case matching with projection, so that the programmer cannot mistakenly project a field from the wrong constructor. We separate projections from cases because it makes the operational semantics simpler, the optimizer simpler, and the proof simpler: our language is an *untyped intermediate language,* not a typed source language. CertiCoq is meant to be used only to compile source programs type-checked in Coq; the Coq type system guarantees that they will not get stuck. Therefore, as the front-end phases are proved correct, the program translated to lower-level intermediate languages (such as the CPS presented here) will not get stuck.

Rule E_APP shows how applications are evaluated. When a function f is applied to arguments $\vec{y}$, we look up f in the environment ρ to retrieve the function value bundle $(\rho', \vec{fd}, f)$. Next, we find function f in $\vec{fd}$ with arguments $\vec{x}$ and function body e. We then evaluate the function body e in saved environment ρ' extended with bindings for each mutually recursive function in $\vec{fd}$ and by associating each y_i in $\vec{y}$ to their respective x_i in $\vec{x}$.

We define set of variables $\mathsf{FV}(e)$ and $\mathsf{BV}(e)$ to be respectively the set of free and bound variables of a term e or of a bundle of function definitions $\vec{fd}$. We also define $\mathsf{names}(\vec{fd})$ to be all the names of functions from the bundle:

$$\mathsf{names}(\vec{fd}) := \{f | f\ (\vec{x})\ =\ e \in \vec{fd}\}$$

An important property that is not enforced by the syntax presented in Fig. 1 is that bound names are globally unique. This property is easy to achieve and maintain; the translation from the previous intermediate language uses a state monad to assign unique variable names. We also make sure that the free variables of the top-level program are disjoint from its bound variables. This allow us, for example, to perform function inlining without worrying about variable capture. We define the proposition $\mathsf{UB}(e)$ to assert that e has the unique binding property.

Applicative context

We define a notion of applicative context, intuitively a term with a hole, which will be used in the statement of the rewriting rules and in the proof of correctness of our function inliner.

$$\begin{array}{llcl}
\text{(Function Context)} & fc & ::= & f\ (\vec{x})\ =\ C \\
\text{(Expression Context)} & C & ::= & [\![\ ]\!] \\
 & & & \mid \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ C \\
 & & & \mid \mathsf{let}\ x = \mathsf{Prim}\ p\ \vec{y}\ \mathsf{in}\ C \\
 & & & \mid \mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ C \\
 & & & \mid \mathsf{let}\ \vec{fd}\ \mathsf{in}\ C \\
 & & & \mid \mathsf{let}\ \vec{fd} + fc :: \vec{fd}\ \mathsf{in}\ e \\
 & & & \mid \mathsf{match}\ x\ \mathsf{with}\ \vec{b} + (c \Rightarrow C) :: \vec{b}
\end{array}$$

Figure 3: Applicative Context

An applicative context is either a hole, a let-binder over an applicative context, a case construct where one of the branches is an applicative context or a function bundle where exactly one of the function bindings has an expression context as body. An expression e can be placed in the hole of a context C to form expression $C[\![e]\!]$. Similarly, a context C_2 can be placed in the hole of a context C_2 to form a composed context $C_1 \cdot C_2$.

We define set $\mathsf{BV}_{\mathsf{stem}}(C)$ to be the variables bound on the stem of C, the variables in scope at the hole in the applicative context C:

$$\mathsf{FV}(C[\![e]\!]) \quad = \quad \mathsf{FV}(C) \cup (\mathsf{FV}(e) \setminus \mathsf{BV}_{\mathsf{stem}}(C))$$

For example,

$$\begin{aligned}
&\mathsf{BV}_{\mathsf{stem}}\Big(\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ \mathsf{let}\ \vec{fd} + (f\ (\vec{z})\ =\ [\![\ ]\!]) :: \vec{fd}'\ \mathsf{in}\ e\Big) \\
&\qquad = \{x,\ f\} \cup \vec{z} \cup \mathsf{names}(\vec{fd} + \vec{fd}')
\end{aligned}$$

Logical relation

Our notion of equivalence reuses a step-indexed logical relation developed by Paraskevopoulou for the proof of correctness of CertiCoq's closure-conversion phase [6]. The main idea is that terms e_1 and e_2 are related at index k ($e_1 \cong_k^{\mathrm{val}} e_2$) whenever they are observationally equal for up to k β-reductions ($e_1 \approx_k e_2$).

Two values v and w are related ($v \cong_k^{\mathrm{val}} w$) if $k = 0$ or if:

- both are constructors with equivalent arguments: $v = c\ v_1\ \dots\ v_n$, $w = c\ w_1\ \dots\ w_n$ and $\forall_{i=1}^{n}, v_i \cong_k^{\text{val}} w_i$
- both are functions, and for any related list of arguments, they evaluate to the related values, which is to say: $v = (\rho_1',\ \vec{fd}_1,\ f_1)$, $w = (\rho_2',\ \vec{fd}_2,\ f_2)$ with $(f_1\ (\vec{x})\ =\ e_1) \in \vec{fd}_1$, $(f_2\ (\vec{y})\ =\ e_2) \in \vec{fd}_2$ and, given two lists $v_1, \dots, v_n$ and $w_1, \dots, w_n$ of related values ($v_i \cong_{k-1}^{\text{val}} w_i$), evaluating the functions' body after extending the functions' environments with these related mappings (for all $f_i \in \vec{fd}_1$, $g_i \in \vec{fd}_2$, $x_i \in \vec{x}$, $y_i \in \vec{y}$, $v_i \in \vec{v}$ and $w_i \in \vec{w}$) produces related values:

$$\begin{array}{ll}(\rho_1'; f_i \mapsto (\rho_1', \vec{fd}_1, f_i); x_i \mapsto v_i,\ e_1) & \cong_{k-1}^{\text{exp}} \\ (\rho_2'; g_i \mapsto (\rho_2', \vec{fd}_2, g_i); y_i \mapsto w_i,\ e_2) & \end{array}$$

Two environments ρ_1 and ρ_2 are related ($\rho_1 \cong_k^{\text{env}} \rho_2$) if, for every variable x, either x is not present in either, or $\rho_1\ x = v_1$ and $\rho_2\ x = v_2$ and $v_1 \cong_k^{\text{val}} v_2$.

Two terms e_1 and e_2 are related under environments ρ_1 and ρ_2 (written $(\rho_1, e_1) \cong_k^{\text{exp}} (\rho_2, e_2)$) if they evaluate to related values. More precisely, they are related at index k if, whenever $\rho_1 \vdash e_1 \Downarrow_j v_1$ (with $j \le k$), then there exists some j' and v_2 such that $\rho_2 \vdash e_2 \Downarrow_{j'} v_2$ and $v_1 \cong_{k-j}^{\text{val}} v_2$.

If, for all i and for all environments ρ_1 and ρ_2 such that $\rho_1 \cong_i^{\text{env}} \rho_2$, two terms e_1 and e_2 are related according to $(\rho_1, e_1) \cong_i^{\text{exp}} (\rho_2, e_2)$, then e_1 and e_2 are contextually equivalent ($e_1 \approx e_2$).

3 GENERAL REWRITES

Figure 4 shows the rules of our general rewriting system, which we then prove correct using the above logical relation.

Dead variables

When a variable does not occur statically within its scope, we can remove its binding without affecting evaluation. Dead variable rewriting rules have the form let $x =$ _ in $e \rightsquigarrow e$, where _ is any let-binding construct in our object language, for example Con $c\ \vec{y}$, whenever x is not used in e.

To handle removal of dead mutually recursive functions, things are a bit more complicated; we use two rules to handle different scenarios under which it is safe to remove function bindings. Dead_bundle, removes a bundle of mutually recursive functions if none of them occurs in the rest of the term. However, this is too coarse-grain to handle the case where only some of the functions in the bundle are dead. For this situation, Dead_fun removes a function definition if it has no applied occurrences outside its own body.

Folding and inlining

Folding and inlining rules perform general reduction steps at compile time. One such folding rule is Fold_case, which performs ι-reduction whenever the correct branch can be statically predicted. Fold_case would be used to perform this reduction:

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{Con}\ S\ y\ \mathsf{in} \\ \quad \mathsf{match}\ x\ \mathsf{with}\ (O \Rightarrow M); (S \Rightarrow \mathsf{let}\ z = \mathsf{Proj}_1\ x\ \mathsf{in}\ N) \\ \rightsquigarrow \mathsf{let}\ x = \mathsf{Con}\ S\ y\ \mathsf{in}\ (\mathsf{let}\ z = \mathsf{Proj}_1\ x\ \mathsf{in}\ N)\end{array}$$

As our language separates pattern-matching into the matching and the binding of projections, we can simply (using firstMatch) return the body of the first branch in $\vec{b}$ matching c in place of the match and let other reductions handle the projections, if any. These projections can then be folded using rule Fold_Proj. It lets us eliminate making a binding y for the nth projection of the value bound to x if x is bound in the context to $c\ \vec{z}$ and the nth variable of $\vec{z}$ is not rebound in the term. For example, we could further reduce the previous example:

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{Con}\ S\ y\ \mathsf{in}\ (\mathsf{let}\ z = \mathsf{Proj}_0\ x\ \mathsf{in}\ N) \\ \rightsquigarrow \mathsf{let}\ x = \mathsf{Con}\ S\ y\ \mathsf{in}\ (z \mapsto y)N\end{array}$$

Function inlining replaces a call to a function by the body of the function. If this was the only call to the function, the function definition can then be eliminated using the Dead_fun rule. This rule is only valid if $\mathsf{FV}(e)$ and the functions' name from $\vec{fd}$, including f, are disjoint from the variables bound on the stem of C, and if the function's parameters $\vec{x}$ and the free variables of e are disjoint from arguments $\vec{y}$, so that there is no variable capture occurring.

General rewrite system

From the rewrite rules shown so far, we create a rewrite system which will describe the transformations that our optimizations apply to a program.

We first take the contextual closure of $\rightsquigarrow$, denoted $\rightsquigarrow_C$, defined as:

$$\frac{e_1 = C[\![e_1']\!] \quad e_2 = C[\![e_2']\!] \quad e_1' \rightsquigarrow e_2'}{e_1 \rightsquigarrow_C e_2}$$

This allows reductions to happen anywhere in the term, following the usual notion of general reductions.

We then take the reflexive transitive closure of the contextual closure of the general rewrite rules to form a system of *General Reduction*, denoted $e \rightsquigarrow_C^* e'$.

Proof of correctness

Our correctness theorem has the following form: Any two terms related by general rewrites evaluate, under equivalent environments, to equivalent values.

Theorem 3.1 (Correctness of GR).

$$\begin{array}{l}\forall e_1\ e_2,\ e_1 \rightsquigarrow_C^* e_2 \implies \\ \quad \forall \rho_1\ \rho_2\ k,\ \rho_1 \cong_k^{\text{env}} \rho_2 \implies (\rho_1, e_1) \cong_k^{\text{exp}} (\rho_2, e_2)\end{array}$$

We prove a generalization of contextual compatibility that allows us to prove nonlocal rewrite rules. Contextual compatibility states that two expressions e_1 and e_2 are related (at k) under a given applicative context C in related evaluation environments ρ_1 and ρ_2 if, for any related ρ_3 and ρ_4, e_1 and e_2 are related (at k). This is because C will affect related ρ_1 and ρ_2 in the same way, resulting in related ρ_3 and ρ_4.

Remark 3.2 (Contextual Compatibility).

$$\begin{array}{l}\forall e_1\ e_2\ C\ \rho_1\ \rho_2\ k, \\ \left(\forall \rho_3\ \rho_4,\ \rho_3 \cong_k^{\text{env}} \rho_4 \implies (\rho_3, e_1) \cong_k^{\text{exp}} (\rho_4, e_2)\right) \implies \\ \quad \rho_1 \cong_k^{\text{env}} \rho_2 \implies \\ \qquad (\rho_1, C[\![e_1]\!]) \cong_k^{\text{exp}} (\rho_2, C[\![e_2]\!])\end{array}$$

$$\frac{f \notin \mathsf{FV}(e) \quad \forall_{(f'\ (\vec{y})\ =\ e_3) \in \vec{fd_1} + \vec{fd_2}}, f \notin \mathsf{FV}(e_3)}{\mathsf{let}\ \vec{fd_1} + (f\ (\vec{x})\ =\ e_1) :: \vec{fd_2}\ \mathsf{in}\ e_2 \rightsquigarrow \mathsf{let}\ \vec{fd_1} + \vec{fd_2}\ \mathsf{in}\ e_2}\ \textsc{Dead\_fun} \qquad \frac{\forall f \in \mathsf{names}(\vec{fd}), f \notin \mathsf{FV}(e)}{\mathsf{let}\ \vec{fd}\ \mathsf{in}\ e \rightsquigarrow e}\ \textsc{Dead\_bundle}$$

$$\frac{x \notin \mathsf{FV}(e)}{\mathsf{let}\ x = \_\ \mathsf{in}\ e \rightsquigarrow e}\ \textsc{Dead\_var} \qquad \frac{(f\ (\vec{x})\ =\ e) \in \vec{fd} \quad (\mathsf{FV}(e) \bigcup \mathsf{names}(\vec{fd})) \bigcap \mathsf{BV}_{\mathsf{stem}}(C) = \emptyset \quad (\mathsf{BV}(e) \bigcup \vec{x}) \bigcap \vec{y} = \emptyset}{\mathsf{let}\ \vec{fd}\ \mathsf{in}\ C[\![\mathsf{App}\ f\ \vec{y}]\!] \rightsquigarrow \mathsf{let}\ \vec{fd}\ \mathsf{in}\ C[\![(\vec{x} \mapsto \vec{y})e]\!]}\ \textsc{Inl\_fun}$$

$$\frac{x \notin \mathsf{BV}_{\mathsf{stem}}(C) \quad (c \Rightarrow e) \in \vec{b}}{\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ C[\![\mathsf{match}\ x\ \mathsf{with}\ \vec{b}]\!] \rightsquigarrow \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ C[\![e]\!]}\ \textsc{Fold\_case}$$

$$\frac{x \notin \mathsf{BV}_{\mathsf{stem}}(C) \quad z_n \notin \mathsf{BV}_{\mathsf{stem}}(C) \bigcup \mathsf{BV}(e)}{\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{z}\ \mathsf{in}\ C[\![\mathsf{let}\ y = \mathsf{Proj}_n\ x\ \mathsf{in}\ e]\!] \rightsquigarrow \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{z}\ \mathsf{in}\ C[\![(y \mapsto z_n)e]\!]}\ \textsc{Fold\_proj}$$

Figure 4: General Rewrite Rules

$$\frac{|e|_x = 0}{\mathsf{let}\ x = \_\ \mathsf{in}\ e \rightharpoonup e}\ \textsc{S\_Dead\_var} \qquad \frac{\forall f \in \mathsf{names}(\vec{fd}),\ |e|_f = 0}{\mathsf{let}\ \vec{fd}\ \mathsf{in}\ e \rightharpoonup e}\ \textsc{S\_Dead\_bundle}$$

$$\frac{|\mathsf{let}\ \vec{fd_1} + \vec{fd_2}\ \mathsf{in}\ Q|_f = 0}{\mathsf{let}\ \vec{fd_1} + (f\ (\vec{x})\ =\ e) :: \vec{fd_2}\ \mathsf{in}\ Q \rightharpoonup \mathsf{let}\ \vec{fd_1} + \vec{fd_2}\ \mathsf{in}\ Q}\ \textsc{S\_Dead\_fun}$$

$$\frac{(c \Rightarrow e) \in \vec{b}}{\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{x}\ \mathsf{in}\ C[\![\mathsf{match}\ x\ \mathsf{with}\ \vec{b}]\!] \rightharpoonup \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{x}\ \mathsf{in}\ C[\![e]\!]}\ \textsc{S\_Fold\_case}$$

$$\frac{}{\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{z}\ \mathsf{in}\ C[\![\mathsf{let}\ y = \mathsf{Proj}_n\ x\ \mathsf{in}\ e]\!] \rightharpoonup \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{z}\ \mathsf{in}\ C[\![(y \mapsto z_n)e]\!]}\ \textsc{S\_Fold\_proj}$$

$$\frac{|\mathsf{let}\ \vec{fd_1} + (f\ (\vec{x})\ =\ e) :: \vec{fd_2}\ \mathsf{in}\ C|_f = 0}{\mathsf{let}\ \vec{fd_1} + (f\ (\vec{x})\ =\ e) :: \vec{fd_2}\ \mathsf{in}\ C[\![\mathsf{App}\ f\ \vec{y}]\!] \rightharpoonup \mathsf{let}\ \vec{fd_1} + \vec{fd_2}\ \mathsf{in}\ C[\![(\vec{x} \mapsto \vec{y})e]\!]}\ \textsc{S\_Shrink\_fun}$$

Figure 5: Shrink Rewrite Rules

However, this is too weak to prove the correctness of nonlocal rules such as Fold_proj, where, for the term that binds the projection to be related when the binding is substituted with the right projection, we need to ensure ρ_3 and ρ_4 still contain the binding of the constructor.

$$\begin{gathered}(\rho_1, \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{z}\ \mathsf{in}\ C[\![\mathsf{let}\ y = \mathsf{Proj}_n\ x\ \mathsf{in}\ e]\!]) \\ \cong_k^{\exp} \\ (\rho_2, \mathsf{let}\ x = \mathsf{Con}\ c\ \vec{z}\ \mathsf{in}\ C[\![(y \mapsto z_n)e]\!])\end{gathered}$$

In order to prove this, we bind x in the context:

$$\begin{gathered}(\rho_1[x \mapsto (c, \vec{v})], C[\![\mathsf{let}\ y = \mathsf{Proj}_n\ x\ \mathsf{in}\ e]\!]) \\ \cong_k^{\exp} \\ (\rho_2[x \mapsto (c, \vec{v})], C[\![(y \mapsto z_n)e]\!])\end{gathered}$$

We cannot apply Contextual Compatibility here, because "let $y = \mathsf{Proj}_n\ x$ in e" and "$(y \mapsto z_n)e$" are only related in contexts that map x to $(c, \vec{v})$ and z_n to v_N, even though x and z_n cannot appear in C due the premise of Fold_proj. So we must be more precise and state that C will only affect the mapping of variables of $\rho_1[x \mapsto (c, \vec{v})]$ and $\rho_2[x \mapsto (c, \vec{v})]$ which are bound on the stem of C. Thus, we can select a set of variables S not bound in C and only consider ρ_3 and ρ_4 that agree with ρ_1 and ρ_2 on variables from S (this is written $\rho \doteq_S \rho'$). In our previous example, we could select $S = \{x, z_n\}$

Theorem 3.3 (Extended Contextual Compatibility).

$$\begin{aligned}&\forall e_1\ e_2\ C\ \rho_1\ \rho_2\ S\ k,\ \ \mathsf{BV}_{\mathsf{stem}}(C) \bigcap S = \emptyset \implies \\ &\Big(\forall \rho_3\ \rho_4,\ \ \rho_1 \doteq_S \rho_3 \implies \rho_2 \doteq_S \rho_4 \implies \\ &\qquad \rho_3 \cong_k^{\mathrm{env}} \rho_4 \implies (\rho_3, e_1) \cong_k^{\exp} (\rho_4, e_2)\Big) \implies \\ &\quad \rho_1 \cong_k^{\mathrm{env}} \rho_2 \implies (\rho_1, C[\![e_1]\!]) \cong_k^{\exp} (\rho_2, C[\![e_2]\!])\end{aligned}$$

4 SHRINK REDUCTIONS

We now turn to a second rewrite system which refines the general rewrite system shown earlier and brings us a step closer to our goal of proving the correctness of our shrink inlining transformation.

Shrink Rewrites. Most of the rules in Fig. 5 are very similar to the General rewrite rules given earlier. We write $|e|_x$ for the number of applied occurences of variable x in expression e. The main difference is that their assumptions are computational, relying on the number of occurrences and (globally) on the unique binder property rather than on sets such as FV and BV. This is an important distinction which will make our life easier in the proof of correspondence to the algorithm. Consider for example S_Fold_proj. Due to the unique binding property, we can drop the assumption that $x \notin \mathsf{BV}_{\mathsf{stem}}(C)$.

Other than the assumptions, the main difference between the two rewrite systems is the use of S_Shrink_fun in place of Inl_fun. Indeed, the latter does not qualify as a shrink reduction as the overall size of the program grows when we inline a function and keep its definition. S_SHRINK_FUN is only applicable when the inlined function has a single applied occurrence. It is admissible (assuming the unique binding property) from INL_FUN followed by DEAD_FUN.

The shrink-rewrite system and its correctness

We take the reflexive transitive closure of the contextual closure of the Shrink-Rewrite Rules presented in Fig. 5 to form a system of *Shrink Rewrites* denoted $\rightharpoonup_C^*$.

We then prove that terms related by shrink reduction are also related by general reduction:

THEOREM 4.1 (GR INCLUDES SR).

$$\forall e_1\ e_2,\ e_1 \rightharpoonup_C^* e_2 \implies e_1 \rightsquigarrow_C^* e_2$$

Moreover, we use the fact that $\rightharpoonup$ is more restrictive than $\rightsquigarrow$ to prove certain properties for any term related by it. For example, shrink reduction preserves the unique binding property (this includes the disjointness of the bound and free variables of the term):

THEOREM 4.2 (SR PRESERVES UB).

$$\forall e_1 e_2,\ e_1 \rightharpoonup_C^* e_2 \land \mathsf{UB}(e_1) \implies \mathsf{UB}(e_2)$$

The set of bound variables does not increase as we shrink a term:

THEOREM 4.3 (SR REDUCES BV).

$$\forall e_1 e_2,\ e_1 \rightharpoonup_C^* e_2 \implies \mathsf{BV}(e_2) \subseteq \mathsf{BV}(e_1)$$

It does not introduce free variables, for example at the top level:

THEOREM 4.4 (SR REDUCES FV).

$$\forall e_1 e_2,\ e_1 \rightharpoonup_C^* e_2 \implies \mathsf{FV}(e_2) \subseteq \mathsf{FV}(e_1)$$

Therefore, closed terms remain closed under shrink reductions.

5 SHRINK INLINER

Function contract, shown in Figure 6, performs shrink reductions in a single pass down and up a program (or top-level expression) P. As contract proceeds down the term e (initially P, then some e such that $\exists C, P = C[\![e]\!]$), we collect in table ρ the functions and constructors which could respectively be inlined and folded.

In addition to the term e currently being transformed, contract maintains four tables:

- $\sigma : var \rightarrow var$, is a delayed renaming substitution (mapping variables to variables) under which e is being considered.
- $\delta : var \rightarrow nat$, tallies the number of occurrences of each variable in the whole program.
- $\rho : var \rightarrow \mathsf{Value}'$, maps function and constructor variables encountered so far (on the stem of C) to their definitions.

$$(\mathsf{Value}')\quad V \quad ::= \quad (c,\ \vec{x})\ \mid\ (\vec{x}, e)$$

- $\theta : var \rightarrow bool$, indicates which functions have been inlined.

δ is updated using functions decreaseOcc $\sigma\ \delta\ \vec{x}$ which decreases by one the count of each variables in $\sigma\vec{x}$ and decreaseCount $\sigma\ \delta\ e$ which decreases $\delta(x)$ by $|\sigma e|_x$, the number of applied occurrences of x in σe.

In the SML/NJ implementation, these maps are implemented using imperative arrays with constant access time. As our compiler is implemented in Gallina, a pure functional language, we instead represent our variables as positive binary numbers and implement

```
contract σ δ ρ θ e = match e with
 | halt x                ⇒ (halt (σx), δ, θ)
 | let x = Prim p y⃗ in e ⇒ if δ(x) = 0
                           then δ ← decreaseOcc δ σ y⃗
                                contract σ δ ρ θ e
                           else (e′, δ, θ) ← contract σ δ ρ θ e
                                if δ(x) = 0
                                then δ ← decreaseOcc δ σ y⃗
                                     (e′, δ, θ)
                                else (let x = Prim p (σy⃗) in e′, δ, θ)
 | let x = Con c y⃗ in e  ⇒ if δ(x) = 0
                           then δ ← decreaseOcc δ σ y⃗
                                contract σ δ ρ θ e
                           else ρ := ρ[x ↦ (c, y⃗)]
                                (e′, δ, θ) ← contract σ δ ρ θ e
                                if δ(x) = 0
                                then δ ← decreaseOcc δ σ y⃗
                                     (e′, δ, θ)
                                else (let x = Con c (σy⃗) in e′, δ, θ)
 | App f y⃗               ⇒ if δ(f) = 1 ∧ ρ f = (x⃗, e)
                           then δ ← inlineCount δ σ f x⃗ y⃗
                                σ := σ[x⃗ ↦ y⃗]
                                θ := θ[f ↦ ⊤]
                                contract σ δ ρ θ e
                           else (App (σf) (σy⃗), δ, θ)
 | let x = Proj_n y in e ⇒ if δ(x) = 0
                           then δ ← decreaseOcc δ σ y
                                contract σ δ ρ θ e
                           else if ρ(σy) = (c, y⃗)
                                then δ ← foldCount δ σ x y⃗_n y
                                     σ := σ[x ↦ (σy⃗_n)]
                                     contract σ δ ρ θ e
                                else (e′, δ, θ) ← contract σ δ ρ θ e
                                     if δ′(x) = 0
                                     then δ ← decreaseOcc δ′ σ y
                                          e′
                                     else (let x = Proj_n (σy) in e′, δ, θ)
 | match v with b⃗        ⇒ if ρ(σv) = (c, y⃗) ∧ (c ⇒ e) ∈ b⃗
                           then δ ← caseCount δ σ b⃗
                                δ ← decreaseOcc δ σ v
                                contract σ δ ρ θ e
                           else (b⃗′, δ, θ) ← contractCase σ δ ρ θ b⃗
                                (match (σv) with b⃗′, δ, θ)
 | let fd⃗ in e           ⇒ (fd⃗′, δ, ρ′) ← preFun σ δ ρ fd⃗
                           (e′, δ, θ) ← contract σ δ ρ′ θ e
                           (fd⃗″, δ, θ) ← postFun σ δ ρ θ fd⃗′
                           (let fd⃗″ in e′, δ, θ)
```

Figure 6: Shrink Inliner Algorithm

maps by *binary tries*, resulting in logarithmic (over the size of the positive number) access time. As shown in Fig. 11, this is still quite fast. Moreover, if one wanted to use constant-access-time impure arrays (a monadic extension to Coq)—thus recovering the original constant access time—our proof of correctness could easily be adapted.

At the top-level, function contract_top calls contract after initializing the maps: σ = id, δ is initialized to have $\delta(x) = |e|_x$ for each variable x appearing in e, ρ is empty and θ maps all variables to $\perp$.

The function contract calls helper functions to process the branches of a pattern-match (contract_branches, see Figure 9) and blocks of recursive functions (preFun and postFun, see Figures 7 and 8).

When encountering a let-bound constructor "let x = Con $c\ \vec{y}$ in e", we first check, by looking up x in δ, if x does not occur in the whole program, in which case we can remove the binding of x and decrease the occurrence count for each variable in $\vec{y}$ under σ. Otherwise, we recursively shrink-reduce e after updating the environment map with the binding $x \mapsto (c, \vec{y})$. On return, we check again if x is dead in the updated counts (i.e., δ). When encountering a let-bound projection "let x = Proj$_n$ y in e", if x is not dead, we look up σy in our environment map ρ to see if we statically know the construct $(c, \vec{y})$ bound to it. If it is, we can remove the binding of x and replace in the rest of e (by extending the renaming σ) all occurrences of x by the nth projection of $\sigma\vec{y}$ and update the count using "foldCount $\delta\ \sigma\ x\ \vec{y}_n\ y$", setting x to 0, increasing the occurence count of $\sigma\vec{y}_n$ by $\delta(x)$ and decreasing $\sigma\vec{y}$ by one.

When converting pattern-matching construct "match v with $\vec{b}$", we first look up v in ρ to see if we know enough about what is bound to it to select the correct branch, which is to say that $\rho(\sigma v) = (c, \vec{y})$ and $(c \Rightarrow e) \in \vec{b}$, and we proceed to shrink-reduce e after adjusting δ to account for the removed occurence of σv and (using caseCount) for the deletion of all other branches. If σv is not known or if no branches match, we recursively shrink-reduce each of the branches using contractCase (see Fig. 9)

When we get to an application "App $f\ \vec{y}$", we first look up σf in the environment map ρ to see if it is a known function $(\vec{x}, e)$ and if this is the only occurrence of σf. In that case, we inline the function and proceed with shrink inlining within its body e after updating the renaming substitution with mappings $x_i \mapsto (\sigma y_i)$ for each x_i, y_i in $\vec{x}, \vec{y}$ and updating the occurrence count with inlineCount $\sigma\ \delta\ f\ \vec{x}\ \vec{y}$, decreasing to 0 all $x_i \in \vec{x}$ and σf and adding $\delta(\vec{x_i}) - 1$ to each $\sigma\vec{y_i}$.

We process a block of mutually recursive functions "let $\vec{fd}$ in e" by first (using function preFun) adding live functions in $\vec{fd}$ to the environment map ρ. We then apply the contract function to e, the rest of the program. We then traverse $\vec{fd}$ a second time with function postFun, this time converting the body of live, noninlined functions. The second traversal uses the initial ρ rather than the one augmented by preFun, such that we don't inline functions within their mutually recursive bundle. The algorithms of each of those pass are given in Figures 7 and 8.

```
preFun σ δ ρ fd⃗₂ = match fd⃗₂ with
    | [ ]                    ⇒ ([ ], δ, ρ)
    | (f (x⃗) = e_b) :: fd⃗₃ ⇒ if δ(f) = 0
                              then δ ← decreaseCount δ σ e_b
                                   preFun σ δ ρ fd⃗₃
                              else (fd⃗′₃, δ, ρ) ← preFun σ δ ρ fd⃗₃
                                   ρ := ρ[f ↦ (x⃗, e_b)]
                                   ((f (x⃗) = e_b) :: fd⃗′₃, δ, ρ)
```

Figure 7: Pre Function Inlining Algorithm

```
postFun σ δ ρ θ fd⃗₂ = match fd⃗₂ with
    | [ ]                    ⇒ ([ ], δ, θ)
    | (f (x⃗) = e_b) :: fd⃗₃ ⇒ if θ(f)
                              then postFun σ δ ρ θ fd⃗₃
                              else if δ(f) = 0
                                then δ ← decreaseCount δ σ e_b
                                     postFun σ δ ρ θ fd⃗₃
                                else (e′_b, δ, θ) ← contract σ δ ρ θ e_b
                                     (fd⃗′₃, δ, θ) ← postFun σ δ ρ θ fd⃗₃
                                     ((f (x⃗) = e′_b) :: fd⃗′₃, δ, θ)
```

Figure 8: Post Function Inlining Algorithm

preFun $\sigma\ \delta\ \rho\ \vec{fd}$ is used on the downwards pass through the term, removing dead functions (and adjusting the occurrence count map δ accordingly) and adding the live ones to the environment ρ.

postFun $\sigma\ \delta\ \rho\ \theta\ \vec{fd}$ processes a block of mutually recursive function $\vec{fd}$ on the upward pass of contract. For each function $f\ (\vec{x}) = e$, we first check if it has been inlined ($\theta(f) = \top$), in which case we simply remove the binding of the function and continue processing the rest of the block (as the count δ has already been adjusted at the inlining points, as shown in Fig. 6). If the function isn't inlined, we check if it is dead ($\delta(f) = 0$), in which case we delete the binding of f, decrease the count of variables occurring in the body of the function e under the renaming substitution σ and continue processing the rest of the block. Finally, if the function is neither dead nor inlined, we apply the shrink inlining algorithm (see Fig. 6) to the body of the function before processing the rest of the block.

```
contractCase σ δ ρ θ b⃗₂ = match b⃗₂ with
    | [ ]               ⇒ ([ ], δ, θ)
    | (c ⇒ e_b) :: b⃗₃ ⇒ (e′_b, δ, θ) ← contract σ δ ρ θ e_b
                          (b⃗′₃, δ, θ) ← contractCase σ δ ρ θ b⃗₃
                          ((c ⇒ e′_b) :: b⃗′₃, δ, θ)
caseCount δ σ b⃗ = match b⃗ with
    | [ ]             ⇒ δ
    | (c′ ⇒ e) :: b⃗ ⇒ if c = c′
                        then decreaseCount δ σ (snd b⃗)
                        else δ ← decreaseCount δ σ e
                             caseCount δ σ b⃗
```

Figure 9: Case Algorithm

Proof of termination

contract is not structurally recursive. While most recursive calls are done on a strictly smaller subterm of its term input e, the inlining case receives a one-AST-node program (App $f\ \vec{y}$) and calls contract on the body of f as found in map ρ. However, if we believe our algorithm is indeed applying shrink reduction to the term, as we are going to prove next, we know that the size of the overall program is decreasing. We can use the other inputs of contract to approximate the size of the whole program. At any point in the algorithm contract e, while converting program P, there exists some applicative context C such that $P = C[\![e]\!]$. This context C consists of all of the bindings encountered on the way to e, some of which (those eligible to be inlined or folded) are reflected in ρ, minus all of the functions which have already been inlined. Our termination measure for contract $\sigma\ \delta\ \rho\ \theta\ e$ is $|e| + |\rho|_\theta$ where $|e|$ is the number of AST nodes in e and $|\rho|_\theta$ the environment map size, defined as:

$$|\rho|_\theta = \sum_{x \in \mathbb{D}(\rho)} \text{if } \left(\rho(x) = (\vec{x}, e) \wedge \theta(x) = \bot\right) \text{ then } |e| \text{ else } 0$$

Which is to say that we add up to the size of e the size of each body of noninlined functions (according to θ) in ρ.

This approximation of the size of P is enough to show termination. For example, in the function inlining case where $\rho(f) = (\vec{x}, e)$ and $\theta(f) = \bot$, we start we size $|\text{App } f\ \vec{y}| + |\rho|_\theta$ and the recursive call has measure $|e| + |\rho|_{\theta[f \mapsto \top]}$ which can easily be shown to be smaller:

$$\begin{aligned} |\text{App } f\ \vec{y}| + |\rho|_\theta &= 1 + |\rho_{\setminus f}|_\theta + |e| \\ &= 1 + |\rho|_{\theta[f \mapsto \top]} + |e| \\ &< |e| + |\rho|_{\theta[f \mapsto \top]} \end{aligned}$$

Termination of the helper functions is proven in a similar manner. For contractCase, we keep track of the fact that the current $\vec{b}$ is a suffix of the original one $\vec{b'}$, and as such for any $(c \Rightarrow e) \in \vec{b}$, $|e| < |\text{match } y \text{ with } \vec{b'}|$, and similarly for postFun with the list of function declaration $\vec{fd}$.

Proof of correspondence

Our proof of correspondence relies on top-level programs being closed. The main theorem for the correspondence of contractTop with our shrink-rewrite system is stated as:

THEOREM 5.1 (contractTop ON CLOSED PROGRAM IS IN SR).

$$\mathsf{UB}(P) \wedge \mathsf{CLO}(P) \implies (P \rightharpoonup_C^* \text{contractTop } P)$$

where $\mathsf{CLO}(e)$ is defined as $\mathsf{FV}(e) = \{\}$.

This composes with Theorem 4.1 and further with Theorem 3.1 to have:

THEOREM 5.2 (CORRECTNESS OF contractTop).

$$\mathsf{UB}(P) \wedge \mathsf{CLO}(P) \implies P \approx \text{contractTop } P$$

We might like to apply the shrink-reducer to open terms as well. For any term P with the unique-binding property $\mathsf{UB}(P)$, there exists a context C such that $\mathsf{CLO}(C[\![P]\!])$ and $\mathsf{UB}(C[\![P]\!])$; where C is constructed such that for any sequence of rewrites $C[\![P]\!] \rightharpoonup_C^* e'$, there exists some e'' such that $e' = C[\![e'']\!]$ and $P \rightharpoonup_C^* e''$. Thus, we can use an alternative function contractTop$'$ which closes term P with C, performs shrink reductions and then returns the unpacked term. For this function, we have:

THEOREM 5.3 (CORRECTNESS OF contractTop$'$).

$$\mathsf{UB}(P) \implies P \approx \text{contractTop}'\ P$$

As we recur down the program P and populate the different maps carried by contract, we need a generalization of this theorem where the current term e being converted is related to the state of the top level program P. Every time the algorithm modifies the term, we have to justify it through our shrink-rewrite rules, which may depend on global properties about the program being transformed. For example, removing the definition of a dead variable involves invoking the DEAD_VAR rule which assume that the variable does not occur in the rest of the program, which would be inconvenient to calculate every time we want to use it. For that reason, a big part of the correspondence proof is to show that the maps that are maintained in the algorithm correctly represent the state of the whole program. Intuitively, while converting program P, at any point in the algorithm where we call contract $\sigma\ \delta\ \rho\ \theta\ e$, there exists some applicative context C such that $P = \sigma(\text{inline } C\ \theta)[\![\sigma e]\!]$, where inline is a function that removes the definition of any function f in C such that $\theta(f) = \top$. P is the state of the program, and each of the maps σ, ρ, δ and θ are correct (according to their invariants) for it. The reductions applied as we process term e affect P and the maps are adjusted accordingly.

The generalized theorem is:

THEOREM 5.4 (contract IS IN SR).

$$\begin{aligned} &\textit{let } P := \sigma(\text{inline } C\ \theta)[\![\sigma e]\!], \\ &\mathsf{UB}(P) \wedge \\ &\mathsf{CLO}(P) \wedge \\ &\mathsf{INV}_P(\delta) \wedge \\ &\mathsf{INV}_C(\rho) \wedge \\ &\mathsf{INV}_{\rho, P}(\theta) \wedge \\ &\mathsf{INV}_{\sigma(\text{inline } C\ \theta), (\sigma e)}(\sigma) \implies \\ &\qquad \exists e'\ \delta_r\ \theta_r, \\ &\qquad \textit{let } P' := \sigma(\text{inline } C\ \theta')[\![\sigma e']\!], \\ &\qquad (e', \delta_r, \theta_r) = \text{contract } \sigma\ \delta\ \rho\ \theta\ e\ \wedge \\ &\qquad P \rightharpoonup_C^* P' \wedge \\ &\qquad \mathsf{INV}_{P'}(\delta_r) \wedge \\ &\qquad \mathsf{INV}_{\rho, P'}(\theta') \\ &\qquad \mathsf{INV}_{\sigma(\text{inline } C\ \theta_r), (\sigma e')}(\sigma). \end{aligned}$$

which is to say that when running contract e with maps respecting their invariants and corresponding to a program P, contract returns a term e' and modified maps δ_r and θ_r describing the updated program P', and proofs that P shrink rewrites to P' and that the invariants still hold on current maps on the new state. The proof goes by induction on the size of the approximation of P given by $|e| + |\rho|_\theta$, just like the proof of termination.

We now detail the invariant on each of the maps and give a sketch of their importance in the proof of correspondence, before describing the auxiliary lemmas to handle case and functions.

$\mathsf{INV}_{C,e}(\sigma)$

σ is a renaming substitution under which the program is being considered. Its invariant states that any variable in its domain is not bound in P, and that variables in its range are either dead or bound on the stem of C:

$$\begin{aligned} \mathsf{INV}_{C,e}(\sigma) := \quad &\forall x\ y, \\ &(x \mapsto y) \in \sigma \implies x \notin \mathsf{BV}(P) \\ &\qquad \wedge |P|_y = 0 \vee y \in \mathsf{BV}_{\text{stem}}(C) \end{aligned}$$

σ is applied everywhere in P, both in e and in C. Due to the unique binding property, adjustment to σ due to a variable bound in e will not affect C, because the variable could not occur free (or otherwise) in C (σweaken). Moreover, the domain of σ is disjoint from its codomain. Combined with the fact that we only add to σ mapping from binding we remove (inlined functions arguments, folded projections, etc.), we can freely fuse multiple delayed substitutions together (σfuse) or stage them as needed, as shown in Figure 10.

$\mathsf{INV}_{\rho,P}(\theta)$

θ keeps track of which functions have been inlined by the algorithm. θ is threaded through the algorithm, and it is shown monotonic, which is to say that for any variable f, if $\theta(f) = \top$ for input θ then output θ_r will have $\theta_r(f) = \top$, which is important to prove termination of contract. For the proof of correctness of contract, θ's invariant states that inlined functions and their arguments do not appear bound in P.

$$\begin{aligned}\mathsf{INV}_{\rho,P}(\theta) := \quad & \forall f,\ \theta(f) = \top \implies \\ & f \notin \mathsf{BV}(P) \land \rho(f) = (\vec{x}, e) \implies \vec{x} \notin \mathsf{BV}(P)\end{aligned}$$

$\mathsf{INV}_P(\delta)$

δ accounts for the number of occurrences of each variable in P. Its invariant is stated as:

$$\mathsf{INV}_P(\delta) := \quad \forall x,\ |P|_x = \delta(x)$$

We say δ is a correct count for P if for all variables x, $\delta(x)$ is exactly the number of times x occurs in P. In the statement of the theorem, this accounts for the delayed substitution σ and for the bodies of inlined functions according to θ.

The unique binding property is important here again to ensure the algorithm updates the count correctly. For example, on "contract let $x = \mathsf{Proj}_n\ y$ in e", we know δ is correct for "$(\sigma C_\theta)[\![\sigma(\mathsf{let}\ x = \mathsf{Proj}_n\ y\ \mathsf{in}\ e)]\!]$" for some C which respects the provided maps. In the case where we fold the projection, we need to prove that δ after "foldCount $\delta\ \sigma\ x\ \vec{y}_n\ y$" is correct for "$(\sigma_{x\mapsto(\sigma\vec{y}_n)}\ C_\theta)[\![\sigma[x \mapsto \sigma\vec{y}_n]e]\!]$". We can first observe that x cannot occur in C_θ due to the unique binding property, so this is equivalent to "$(\sigma C_\theta)[\![\sigma[x \mapsto (\sigma\vec{y}_n)]e]\!]$". By the invariant on σ, we know that x is neither in the domain or the range of σ and as such "$\sigma[x \mapsto (\sigma\vec{y}_n)]e$" is the same as "$(x \mapsto (\sigma\vec{y}_n))(\sigma e)$", which is to say we can first apply σ before substituting $\sigma\vec{y}_n$ for x. Finally, by the unique binding property, we know that x will not be bound in e such that all of its occurrences will be replaced by $\sigma\vec{y}_n$, which brings us to the correct count.

$\mathsf{INV}_C(\rho)$

ρ is a view of the current context. The invariant for ρ asserts that it contains every function and constructor on the stem of C and nothing more.

$$\begin{aligned}\mathsf{INV}_C(\rho) := \quad & \forall x, \\ & \rho(x) = (c, \vec{y}) \leftrightarrow \exists C_1 C_2, C = C_1 \cdot (\mathsf{let}\ x = \mathsf{Con}\ c\ \vec{y}\ \mathsf{in}\ C_2) \\ & \land \\ & \rho(x) = (\vec{y}, e) \leftrightarrow \exists C_1 C_2\ \vec{fd}, C = C_1 \cdot (\mathsf{let}\ \vec{fd}\ \mathsf{in}\ \ C_2) \\ & \qquad\qquad \land (x\ (\vec{y})\ =\ e) \in \vec{fd}\end{aligned}$$

Some of the function in ρ may have been inlined (such that they are not in inline $C\ \theta$) and are thus not eligible to be inlined. However, this means they do not occur in P, so we will will never look them up in ρ again.

Auxiliary proofs

When converting case and bundles of functions, we call the auxiliary functions shown in figure 9, 7 and 8. Just like for contract, we need to carefully select a P that best represents the current state of the program; it is important to be aware of which portions of the term have already been converted as they no longer need to be considered under delayed σ and what is available to be folded or inlined.

Case. When contracting term "match x with $\vec{b}$", we first verify if we can fold the statement. If this is not possible, we contract each of the branches in $\vec{b}$ using function contractCase. As we progress through the lists of branches, $\vec{b}$ is split into the contracted branches $\vec{b_1}$ (initially empty) and its remaining suffix $\vec{b_2}$ (empty when the contractCase returns to contract) . When calling "contractCase $\sigma\ \delta\ \rho\ \theta\ \vec{b_2}$", the current state P is

$$\sigma(\mathsf{inline}\ C\ \theta)[\![\mathsf{match}\ x\ \mathsf{with}\ (\vec{b_1} + \sigma\vec{b_2})]\!]$$

The invariant on σ allows us to prove that $x = \sigma x$ and $\vec{b_1} = \sigma\vec{b_1}$ such that

$$\mathsf{match}\ x\ \mathsf{with}\ (\vec{b_1} + \sigma\vec{b_2}) = \sigma\Big(\mathsf{match}\ x\ \mathsf{with}\ (\vec{b_1} + \vec{b_2})\Big)$$

On $b_2 = (c \Rightarrow e_b) + b_3$, we can rewrite the state as

$$\sigma\Big(\mathsf{inline}\ \Big(C \cdot \mathsf{match}\ x\ \mathsf{with}\ \vec{b_1} + (c \Rightarrow [\![\ ]\!]) :: \vec{b_3}\Big)\ \theta\Big)[\![\sigma e_b]\!]$$

to recur on e_b with contract. On return b_3' with updated δ_r and θ_r, the state is

$$P' = \sigma(\mathsf{inline}\ C\ \theta_r)[\![\mathsf{match}\ x\ \mathsf{with}\ \vec{b_1} + (c \Rightarrow e_b') :: \vec{b_3}]\!]$$

We also return proofs that $P \rightharpoondown_C P'$, that δ_r is a correct count for P', that the invariant for θ_r holds for σ and P' and that the invariant for σ holds for P'.

Functions. When contract is called on a bundle of functions "let $\vec{fd}$ in e", we first call "preFun $\sigma\ \delta\ \rho\ \vec{fd}$", before converting e and calling "postFun $\sigma\ \delta\ \rho\ \theta\ \vec{fd}$". The carried maps already account for some prefix $\vec{fd_1}$ for which $\vec{fd_1} + \vec{fd_2} = \vec{fd}$, with $\vec{fd_1} = [\ ]$ at first.

For "$\vec{fd_2'} \leftarrow$ preFun $\sigma\ \delta\ \rho\ \vec{fd_2}$", program P, originally

$$\sigma(\mathsf{inline}\ C\ \theta)[\![\sigma(\mathsf{let}\ \vec{fd_1} + \vec{fd_2}\ \mathsf{in}\ e)]\!]$$

is updated to

$$P' = \sigma(\mathsf{inline}\ C\ \theta)[\![\sigma(\mathsf{let}\ \vec{fd_1} + \vec{fd_2'}\ \mathsf{in}\ e)]\!]$$

with $P \rightharpoondown_C P'$. Functions in $\vec{fd_2}$ which are already dead have their bindings removed from $\vec{fd_2'}$. δ_r is updated accordingly, and is correct from P'. The updated environment ρ_r adds to ρ all the functions bound by $\vec{fd_2'}$. Because the names in $\vec{fd}$ are disjoint from the inlined functions as tallied by θ, the resulting P' (where $\vec{fd_1}$ is empty) can be rewritten as

$$\sigma\Big(\mathsf{inline}\ (C \cdot \mathsf{let}\ \vec{fd_2'}\ \mathsf{in}\ [\![\ ]\!])\ \theta\Big)[\![\sigma e]\!]$$

which is in the right form to contract e.

When we call "postFun $\sigma\ \delta\ \rho\ \theta\ \vec{fd}$" from contract, e has already been converted by the main function, and we turn on to processing

$$\begin{aligned}
(\sigma C)[\![\sigma(\text{let } x = \text{Proj}_2\ y \text{ in } e')]\!] &= (\sigma C)[\![\text{let } x = \text{Proj}_2\ (\sigma y) \text{ in } (\sigma e')]\!] && \text{by definition}\\
&= (\sigma C)[\![(x \mapsto (\sigma y_2))\ (\sigma e')]\!] && \text{by fold_proj}(\rightharpoonup)\\
&= (\sigma C)[\![(\sigma[x \mapsto (\sigma y_2)])\ e']\!] && \text{by } \sigma\text{fuse}\\
&= (\sigma[x \mapsto (\sigma y_2)])\ C[\![(\sigma[x \mapsto (\sigma y_2)])\ e']\!] && \text{by } \sigma\text{weaken}
\end{aligned}$$

Figure 10: Example of substitution fusion

the bodies of live, noninlined functions in $\vec{fd}$. After converting e, the program state P is

$$\sigma\Big(\text{inline } (C \cdot \text{let } \vec{fd_1} + \vec{fd_2} \text{ in } [\![\ ]\!])\ \theta\Big)[\![e_r]\!]$$

When $\vec{fd_2} = (f\ (\vec{x}) \ = \ e_b; \vec{fd_3})$ for some live f, we need to show that we can rewrite P to be of the right form for its body e_b to be translated (into e'_b) using convert:

$$\sigma\Big(\text{inline } \big(C \cdot \text{let } \vec{fd_1} + (f\ (\vec{x}) \ = \ e_b) :: \vec{fd_3} \text{ in } [\![\ ]\!]\big)\ \theta\Big)[\![e_r]\!]$$
$$=$$
$$\sigma\Big(\text{inline } \big(C \cdot \text{let } \vec{fd_1} + (f\ (\vec{x}) \ = \ [\![\ ]\!]) :: \vec{fd_3}) \text{ in } e_r\big)\ \theta\Big)[\![e_b]\!]$$

By the invariant on σ and θ, we know e_r is equivalent to σe_r and that inline with θ has no effect on e_r. The proof of correctness for postcontract carries this fact along to be able to move e_r in and out of the context as we recur on functions' bodies. postFun updates δ_r and θ_r and returns $\vec{fd}'_3$ which can form $\vec{fd}'_2 = (f\ (\vec{x}) \ = \ e'_b; \vec{fd}'_3)$, with the resulting state being

$$P' = \sigma\Big(\text{inline } (C \cdot \text{let } \vec{fd_1} + \vec{fd}'_2 \text{ in } [\![\ ]\!])\ \theta_r\Big)[\![e_r]\!]$$

which can be rewritten as

$$\sigma(\text{inline } C\ \theta_r)[\![\text{let } \vec{fd_1} + \vec{fd}'_2 \text{ in } e_r]\!]$$

We also return proofs that $P \rightharpoonup_C P'$ and that the maps properly characterize P'.

6 PERFORMANCE MEASUREMENTS

We have tested the effect of the shrink inliner on a few programs when evaluated in the intermediate language on which the transformation is performed. The results are included in Figure 11. *Binom* is an implementation of binomial queues [7] (priority queues with log-time insert, delete-min, and merge). *Color* runs a verified implementation of the Kempe/Chaitin algorithm for graph coloring [8] on a large graph. *Veristar* [9] is a verified theorem prover (resolution theorem proving with paramodulation) for a subset of separation logic, run over a large entailment.

We see a significant number of functions inlined in a single shrink inlining (S.I.) pass, resulting in substantially smaller programs that run 5x faster. Most of the inlined functions are administrative redexes. Although one pass does not always reduce to shrink-normal form, very few redexes remain for the second and third passes; this justifies the *quasilinear time* designation [1] (actually, for our implementation, *quasi-N* $\log N$ time).

Shrink-inlining is fast: even on a large program such as *Veristar*, it takes a fraction of a second. The table shows that it's important to shrink-inline both before and after closure-conversion; if not run before, closure-conversion takes too long; if not after, the compiled program will run slower.

7 REDUCTION OF ADMINISTRATIVE REDEXES

Administrative redexes are β-redexes introduced by the CPS transformation and that can safely be reduced without affecting the original term. For example, an early CPS transformation [4] converts the term "$(\lambda x.x)\ y$" as

$$\lambda k_1.(\lambda k_2.k_2(\lambda x.\lambda k_3.k_3\ x))(\lambda m.(\lambda k_4.k_4\ y)(\lambda n.(m\ n)\ k_1))$$

Implementations of the CPS transformation in several compilers, in order to generate smaller terms that leave less work for later optimization phases to do, cleverly avoid producing so many administrative redexes [10, 11]. Danvy and Nielsen [12] give a comprehensive account of different CPS transformations and on the administrative redexes they introduce.

But these clever CPS transformations that avoid producing administrative redexes are more difficult to prove correct [5]. Furthermore, some administrative redexes *should not* be reduced! They represent join points of the control flow; reducing them duplicates the instructions following the join point [13]. This duplication occurs in many optimizing CPS transformations over languages with pattern-matching [5, 13].

We recommend: use a simple CPS transformation that makes no effort to reduce administrative redexes; then use shrink-reduction. This is approximately as efficient as the more clever CPS transformation, and it reduces just the right set of redexes, including all the administrative that are not join points.

Theorem 7.1. *All administrative redexes with a single applied occurence will be reduced in a single pass of the shrink inliner.*

The proof is a corollary of our proof of correspondence of the shrink inliner (Theorem 5.4), where we prove that the algorithm correctly tabulates the number of occurrences for every variables in the program, such that administrative redexes with a single applied occurence will be eligible for inlining when we get to them during the first shrink inlining pass.

8 RELATED WORK

The shrink inliner we present in Figure 6 is taken almost directly from Appel and Jim [1], who describe the algorithm implemented in the SML/NJ compiler. They present a set of rewriting rules which was the main source of inspiration for our shrink-rewrite system, and prove its confluence. The main difference is that their algorithm allowed occurrence-counts to be over-approximations, and they split the occurrence-counts into applied and escaping in order to tolerate this approximation. However, with a few changes to the occurrence updates, we can get the exact number of occurrence in our map δ, and as such have no reason to split it into the two type of occurrences.

Benchmark	Binom	Color	Veristar
Size without Shrink Inlining (AST nodes)	3156	76.6k	82.0k
Size with S.I. (AST nodes)	616	28.5k	14.8k
# of evaluation steps without S.I.	4560	120.3M	348.3M
# of evaluation steps with S.I.	1132	26.9M	82.9M
Time for one S.I. pass (sec., running extracted in Ocaml)	0.0069	0.34	0.27
# inlined functions in one S.I. pass	620	9240	14305
# of cases folded by one S.I. pass	2	1	8
# of projections folded by one S.I. pass	2	2	14
# of dead constructors removed by one S.I. pass	41	52	486
# of dead functions removed by one S.I. pass	0	87	51
# of shrink reductions performed by second S.I. pass	0	24	16
# of shrink reductions performed by third S.I. pass	0	3	0
Size after closure conversion without S.I. (AST nodes)	6390	188.5k	255.8k
Size after closure conversion with S.I. (AST nodes)	1163	34.9k	32.3k
Time for closure conversion without S.I. (s.)	0.30	1039.68	481.43
Time for closure conversion with S.I. (s.)	0.0080	3.28	1.41
# of functions inlined by S.I. after closure conversion	0	0	0
# of cases folded by S.I. after closure conversion	0	0	0
# of projections folded by S.I. after closure conversion	6	136	250
# of dead constructors by S.I. removed after closure conversion	4	1261	796
# of dead functions by S.I. removed after closure conversion	0	4	0

Figure 11: Performance measurements

Appel and Jim [1] present a second algorithm that reduces all shrink redexes in linear time using a term representation that heavily uses imperative ref-update. Kennedy [3] further refines this algorithm. Neither Appel and Jim nor Kennedy prove the correctness of any of their algorithms with repect to an evaluation relation.

The cakeML project [14] includes a verified compiler for a "substantial subset of Standard ML". It contains a two-pass optimization inlining small, non-recursive functions. Inlining decisions are not updated as inlining is performed, making it similar to the naive algorithm that Appel and Jim show performs much worse than linear time. The optimization pipeline also includes a constant propagation and folding phase which, for example, folds *if*-statements if their guard can be computed statically. However, doing these optimizations in different phases misses cascading reductions where further optimizations are enabled by each reduction.

Pilsner [15] is a verified compiler with an ML-like source language and CPS-based intermediate language. It includes a simple function-inlining optimization which does not update its inlining decisions during the inlining pass, nor does it inline within the body of inlined functions. It has a dead-variable-elimination phase deleting dead definition in a single pass up and down a program. It addition to missing cascading reductions, we find that dead definitions often arise from other optimizations such as projection-folding which are not currently present in the optimization pipeline of this compiler.

CompCert [16] is a verified optimizing compiler for C. It includes a function inlining pass. However, the decisions to inline are taken in a different pass and are not updated as inlining is done. There is no attempt (and in a C compiler, less need) to combine inlining, constant folding, and dead-variable elimination into a single efficient pass.

Administrative redexes in the context of CPS transformations have been the subject of many papers since being introduced by Plotkin [4]. Our pipeline which consists of a simple CPS transformation followed by a pass which reduces administrative redexes is similar to the two-pass CPS transformation presented by Sabry and Felleisen [11]. However, our shrink inlining pass is not limited to reducing administrative redexes; it also performs case-folding, dead-variable elimination, and reduction of many nonadministrative redexes.

9 CONCLUSION

We have presented a proof of correctness for a shrink inliner compilation phase combining constant folding, function inlining and dead-variable elimination. The full proof composes multiple correspondence proofs, step-by-step refining a semantic notion of equivalence into our syntax-driven algorithm.

We believe other transformations could reuse the proof of correctness of the general rewriting system, either directly or through a refined system such as shrink rewrites.

Proving correspondence of the algorithm to the shrink-rewrite system rather than the general one or the logical relation significantly simplifies the reasoning. As previously stated, some of the invariants on the terms and maps, such as closedness, are preserved by shrink reductions, and as such do not have to be threaded through the proof. Moreover, the shrink-rewrite system already incorporates some optimizations that make it easier to prove the algorithm correspondence. For example, substitution is performed in a global way since the unique binding property prevents any shadowing and capture of variables. Meanwhile, the notion of substitution used for the rewrite system is the more usual one which corresponds closely with the semantics of our language which is defined for nonuniquely bound terms.

Acknowledgments. We thank Abhishek Anand, Trevor Jim, Steve Zdancewic and the anonymous reviewers for valuable feedback. This work was supported in part by NSF grants CCF-1407794 and CCF-1521602.

REFERENCES

[1] Andrew W. Appel and Trevor Jim. Shrinking Lambda Expressions in Linear Time. *J. Funct. Program.*, 7(5):515–540, September 1997.

[2] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. Certicoq: A verified compiler for Coq. In *CoqPL 2017: The Third International Workshop on Coq for Programming Languages*, January 2017. URL http://conf.researchr.org/event/CoqPL-2017/main-certicoq-a-verified-compiler-for-coq.

[3] Andrew Kennedy. Compiling with Continuations, Continued. *SIGPLAN Not.*, 42 (9):177–190, 2007.

[4] Gordon D. Plotkin. Call-by-name, call-by-value and the Îż-calculus. *Theoretical Computer Science*, 1(2):125 – 159, 1975.

[5] Zaynah Dargaye and Xavier Leroy. *Mechanized Verification of CPS Transformations*, pages 211–225. Springer, 2007.

[6] Zoe Paraskevopoulou and Andrew W. Appel. Modular closure conversion, proved correct in Coq. in preparation, 2017.

[7] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, April 1978. ISSN 0001-0782. doi: 10.1145/359460.359478. URL http://doi.acm.org/10.1145/359460.359478.

[8] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[9] Gordon Stewart, Lennart Beringer, and Andrew W. Appel. Verified heap theorem prover by paramodulation. *SIGPLAN Not.*, 47(9):3–14, 2012.

[10] Olivier Danvy and Andrzej Filinski. Representing control: a study of the cps transformation, 1992.

[11] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Conference on LISP and Functional Programming*, pages 288–298, 1992.

[12] Olivier Danvy and Lasse R. Nielsen. CPS Transformation of Beta-redexes. *Inf. Process. Lett.*, 94(5):217–224, June 2005.

[13] Amr A. Sabry. The formal relationship between direct and continuation-passing style optimizing compilers: A synthesis of two paradigms, 1994.

[14] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A New Verified Compiler Backend for CakeML. *SIGPLAN Not.*, 51(9):60–73, 2016.

[15] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A Compositionally Verified Compiler for a Higher-order Imperative Language. *SIGPLAN Not.*, 50(9):166–178, 2015.

[16] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, pages 107–115, 2009.