Deriving Efficient Program Transformations from Rewrite Rules

JOHN M. LI, Princeton University, USA
ANDREW W. APPEL, Princeton University, USA

An efficient optimizing compiler can perform many cascading rewrites in a single pass, using auxiliary data structures such as variable binding maps, delayed substitutions, and occurrence counts. Such optimizers often perform transformations according to relatively simple rewrite rules, but the subtle interactions between the data structures needed for efficiency make them tricky to write and trickier to prove correct. We present a system for semi-automatically deriving both an efficient program transformation and its correctness proof from a list of rewrite rules and specifications of the auxiliary data structures it requires. Dependent types ensure that the holes left behind by our system (for the user to fill in) are filled in correctly, allowing the user low-level control over the implementation without having to worry about getting it wrong. We implemented our system in Coq (though it could be implemented in other logics as well), and used it to write optimization passes that perform uncurrying, inlining, dead code elimination, and static evaluation of case expressions and record projections. The generated implementations are sometimes faster, and at most 40% slower, than hand-written counterparts on a small set of benchmarks; in some cases, they require significantly less code to write and prove correct.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**.

Additional Key Words and Phrases: compiler correctness, compiler optimization, metaprogramming, domain-specific languages, interactive theorem proving, shrink reduction

ACM Reference Format:
John M. Li and Andrew W. Appel. 2021. Deriving Efficient Program Transformations from Rewrite Rules. *Proc. ACM Program. Lang.* 5, ICFP, Article 74 (August 2021), 29 pages. https://doi.org/10.1145/3473579

1 INTRODUCTION

Program transformations are the heart of any good optimizing compiler—so they should be correct, efficient, and easy to write. Functional languages make it easy to describe transformations as rewrite rules, implemented by pattern-matching. But those naive implementations can be unacceptably inefficient. They may fail to get rid of all redexes in one pass, or they may call to helper functions which require an unacceptable number of extra traversals.

For efficiency, compiler writers employ a wide range of implementation tricks, such as delayed substitutions, occurrence counts, and variable binding maps [Appel and Jim 1997; Benton et al. 2004; Kennedy 2007; Peyton Jones and Marlow 2002]. Such implementations can be very efficient indeed, but much harder to write correctly and reason about, as they require complex bookkeeping to properly update auxiliary data structures at every recursive call.

To regain confidence in an implementation, one can *prove* that it behaves as expected. This approach has been taken by several compiler implementations in recent years [Anand et al. 2017;

Authors' addresses: John M. Li, Princeton University, Princeton, USA, johnli@princeton.edu; Andrew W. Appel, Princeton University, Princeton, USA, appel@princeton.edu.

2475-1421/2021/8-ART74
https://doi.org/10.1145/3473579

Kumar et al. 2014; Leroy et al. 2012; Neis et al. 2015], and is highly effective [Yang et al. 2011]. But such proofs are exceptionally difficult. An efficient program transformation, already tricky to reason about on paper, is even more difficult to work with in a formal proof system. Even for a single transformation, articulating the invariants on auxiliary data structures in formal language and using them to prove an implementation correct can be challenging enough to be considered a research contribution on its own [Savary Bélanger and Appel 2017].

In this paper, we describe a method for semi-automatically deriving correct and efficient implementations from specifications:

- by semi-automatic, we mean that large parts of implementation and proof can be automatically generated from a program transformation's specification;
- by correct, we mean that the derived implementation is guaranteed to perform transformations according to a given system of syntactic rewrite rules;
- by efficient, we mean that derived implementations are about as fast as well-optimized hand-written code.

Importantly, specifications of program transformations include not only a high-level description of the transformation to perform, but also a description of the auxiliary data structures needed to implement it efficiently. Our method for deriving implementation and proof from these specifications is then designed around the observation that an efficient program transformation resembles a traversal with zipper [Huet 1997] where expensive computations are incrementalized through the use of extra parameters and state variables.

We have implemented this method as a metaprogram in Coq. It takes a specification which includes a list of rewrite rules to implement, produces a partial functional program with partial proof that its inputs and outputs are related to each other under the given rewrite system, and leaves behind proof obligations corresponding to components whose implementation and proof need guidance from the user. Dependent types ensure that each such component is written correctly. Once all obligations have been solved, one obtains an efficient functional implementation proved correct with respect to the given rewrite rules.[1]

Finally, our tool does not require the user to learn a new domain specific language or framework: the specifications the metaprogram requires can be expressed in terms of ordinary Coq inductive relations, and the proof obligations it generates can be solved using ordinary Coq tactics.

Contributions:

- We develop a theory of efficient program transformation using context-dependent rewrites.
- We show how to derive efficient functional programs from specifications written in this theory.
- We demonstrate a tool to do this mostly automatically.
- We use our tool to implement and verify functional-language optimizations. In some cases, it takes a lot less code (and proof), and benchmarks suggest that the optimizers are about as fast as (i.e. at most 40% slower than) hand-written counterparts.

[1]To obtain a full compiler correctness theorem, one then needs to prove each rewrite rule sound with respect to the semantics of the object language. Such proofs are beyond the scope of our tool. However, we find that when proving program transformations correct by hand, the proofs are often naturally structured in two layers: one that relates the implementation to a set of rewrite rules, and one that proves the rewrite rules semantically sound. The first layer is a substantial part of the entire proof, and our tool helps to automate it away.

Variables	Var	$\ni$	$x, y, \ldots$
Constructor tags	Tag	$\ni$	c
Primitives	Prim	$\ni$	p
Expressions	e	$::=$	$\mathbf{halt}\ x$
		$\mid$	$f(\vec{x})$
		$\mid$	$\mathbf{let}\ x = \mathrm{Con}\ c\ \vec{y}\ \mathbf{in}\ e$
		$\mid$	$\mathbf{let}\ x = \mathrm{Proj}_n\ y\ \mathbf{in}\ e$
		$\mid$	$\mathbf{case}\ x\ \mathbf{of}\ \overrightarrow{c \Rightarrow e}$
		$\mid$	$\mathbf{let}\ x = \mathrm{Prim}\ p\ \vec{y}\ \mathbf{in}\ e$
		$\mid$	$\mathbf{let}\ \overrightarrow{f(\vec{x}) = e}\ \mathbf{in}\ e$

Fig. 1. λ_{cps} syntax. $\overrightarrow{f(\vec{x}) = e}$ is a block of mutually recursive functions, each with 0 or more parameters.

2 WHAT MAKES PROGRAM TRANSFORMATIONS DIFFICULT TO WRITE?

To illustrate how tricky it can be to write efficient program transformations correctly by hand, this section presents some program transformations used by functional-language compilers and the tricks used to implement them efficiently. We have used our tool to implement each of these transformations with a fraction of the effort required to do so manually. Because we specify program transformations in a more principled manner, our tool is able to fill in most of the implementation automatically and generates proof obligations for the remaining parts that prevent subtle interactions between efficiency tricks from introducing bugs.

The user of our tool can provide any first-order Coq inductive type representing an intermediate language, but the examples in this paper will focus on a continuation-passing-style intermediate representation that we call λ_{cps}. The syntax for λ_{cps} terms is given in figure 1. It's an untyped call-by-value lambda calculus with constructors and recursive functions; every call is a tail call.

Although typed ML-like source languages tie constructor-matching and record-projection together in a **case** or **match** expression, λ_{cps} is a low-level untyped language, designed for ease of analysis, optimization, and code generation—so our a case expression just selects the branch with the matching constructor tag. Then each branch must use projections to access constructor arguments.

Figure 2 gives a representation of λ_{cps} as a Coq inductive type. This representation additionally contains fun_tags, which can be used to associate metadata with each function definition and call site.

2.1 Case Folding

The case folding transformation states that if a variable x is bound to a constructor application $c\ \vec{y}$, then any case expression that scrutinizes x later on can be simplified to the branch corresponding to c. As a rewrite rule:

$$\frac{(c' \Rightarrow e') \in \overrightarrow{c \Rightarrow e} \qquad C = D \circ (\mathbf{let}\ x = \mathrm{Con}\ c'\ \vec{y}\ \mathbf{in}\ \square) \circ E \qquad x \text{ not bound on the stem of } E}{C[\mathbf{case}\ x\ \mathbf{of}\ \overrightarrow{c \Rightarrow e}] \rightsquigarrow C[e']}$$

where C, D, and E denote one-hole contexts, $\square$ is the empty context, square brackets denote context application, and $\circ$ denotes context composition.

An efficient implementation carries along a finite map ρ as an extra parameter that associates variable names with constructor applications. Upon encountering a binding $\mathbf{let}\ x = \mathrm{Con}\ c\ \vec{y}\ \mathbf{in}\ e$, we extend ρ with $x \mapsto c\ \vec{y}$ before making a recursive call on e. Upon encountering a case expression

```
Definition var := positive.
Definition fun_tag := positive.
Definition ctor_tag := positive.
Definition prim := positive.
Inductive exp: Type := Econstr (x: var) (c: ctor_tag) (ys: list var) (e: exp)
                     | Ecase (x: var) (ces: list (ctor_tag * exp))
                     | Eproj (x: var) (c: ctor_tag) (n: N) (y: var) (e: exp)
                     | Efun (fds: list fundef) (e: exp)
                     | Eapp (f: var) (ft: fun_tag) (xs: list var)
                     | Eprim (x: var) (p: prim) (ys: list var) (e: exp)
                     | Ehalt (x: var)
with fundef: Type := Ffun (f: var) (ft: fun_tag) (xs: list var) (e: exp).
```

Fig. 2. λ_{cps} represented as a Coq inductive type. This type is *not* built into our tool; instead, the user can choose any such type, representing the user's intermediate language.

case x **of** $\overrightarrow{c \Rightarrow e}$, we check if $(x \mapsto c'\ \vec{y}) \in \rho$ and if there exists a case arm $c' \Rightarrow e'$ in the list of case arms $\overrightarrow{c \Rightarrow e}$. If so, we make a recursive call on e', getting rid of the surrounding case expression and all untaken branches in the process.

When a binding of variable x can occur inside the scope of another binding of x, one must delete bindings in ρ to handle this name shadowing. We avoid this problem by enforcing a global invariant that all bindings are globally unique names, as do many compilers [Appel and MacQueen 1991; Chambart et al. 2016; Kranz et al. 1986; Steele 1978], including all SSA-based compilers [Cytron et al. 1991]. Other compilers enforce a no-shadowing invariant [Peyton Jones and Marlow 2002]; our theory and tool can accommodate either approach.

2.2 Projection Folding

The projection folding transformation states that if a variable x is bound to a constructor application $c\ \vec{y}$ and later we project the nth component from x and bind the result to z, then we can get rid of the projection and instead just use y_n (that is, the nth component of y) in place of z. As a rewrite rule:

$$\frac{C = D \circ (\textbf{let}\ x = \text{Con}\ c\ \vec{y}\ \textbf{in}\ \square) \circ E \qquad x \text{ not bound on the stem of } E}{C[\textbf{let}\ z = \text{Proj}_n\ x\ \textbf{in}\ e] \rightsquigarrow C[e[y_n/z]]}$$

An efficient implementation carries along a finite map ρ, as in case folding, to associate names x with constructor applications $c\ \vec{y}$. Calling a helper function to perform the substitution $e[y_n/x]$ would make the transformation take quadratic time. To achieve quasilinear runtime, we use an extra parameter σ to represent a delayed substitution. This substitution is applied to each free variable throughout the traversal. Upon encountering a projection **let** $z = \text{Proj}_n\ x$ **in** e, we check whether $(\sigma x \mapsto c\ \vec{y}) \in \rho$; if so, we drop the projection and make a recursive call on e with σ extended by $(z \mapsto y_n)$. To avoid issues around shadowing, this implementation strategy maintains the global-unique-bindings invariant.

This optimization is often done simultaneously with case folding. The two optimizations interact in a nontrivial way: upon encountering a case expression **case** x **of** $\overrightarrow{c \Rightarrow e}$, one must now check whether σx occurs in ρ instead of just x.

2.3 Dead Function Elimination

Dead function elimination removes definitions of nonrecursive functions that are never used. As a rewrite rule:

$$\frac{f \notin \mathrm{FV}(e_2)}{C[\textbf{let } f(\vec{x}) = e_1 \textbf{ in } e_2] \rightsquigarrow C[e_2]}$$

A naive implementation might call a helper function to compute $\mathrm{FV}(e_2)$ in order to check whether a function f is dead or not, requiring an extra traversal and the construction of a free variable set.

One possible efficient implementation could be to return both a transformed expression and its free variable set. Then, upon encountering a function definition $\textbf{let } f(\vec{x}) = e_1 \textbf{ in } e_2$, we could first make a recursive call on e_2 to obtain a transformed expression e_2' along with its free variables $\mathrm{FV}(e_2')$. If $f \notin \mathrm{FV}(e_2')$ then we could simply drop the definition and return $(e_2', \mathrm{FV}(e_2'))$; otherwise, we could recursively transform the function body.

This is asymptotically faster than a naive implementation using a helper function, but still has a problem. Suppose we would like to perform dead variable elimination simultaneously with inlining. Then, upon encountering a function definition $\textbf{let } f(\vec{x}) = e_1 \textbf{ in } e_2$, we have the following dilemma:

- If f is inlinable multiple times in e_2, then we would like to first make a recursive call on e_1 so that f can be optimized before it gets inlined.
- If f is dead, then we would like to just drop its definition immediately instead of wasting time optimizing its body.

In other words, we would like to know whether $f \in \mathrm{FV}(e_2)$ even before making any recursive calls. We can do this by maintaining globally unique bindings and a mapping δ from variable names to the number of times they occur in nonbinding position in the entire term being transformed. A function f is dead if $\delta(f) = 0$. Thus, upon encountering a function definition $\textbf{let } f(\vec{x}) = e_1 \textbf{ in } e_2$, we can first use δ to check whether f is dead and behave accordingly. In the case where f is dead, we must maintain the invariant on δ by decrementing the occurrence counts of every variable that occurs in the function body e_1 before deleting it.

In practice, it's desirable to perform dead function elimination simultaneously with case and projection folding. This is because the rewrites can cascade: for example, case folding may remove all branches in which a function f is called, leaving it dead; removing f's definition may leave even more definitions dead, and so on. Note again that, when doing all three of these at once, their respective implementation strategies interact in subtle ways. For example, now case folding must traverse all branches scheduled for deletion in order to decrement occurrence counts properly; morever, this decrementing operation must properly apply projection folding's delayed substitution σ along the way.

2.4 Uncurrying

The following function f is a curried function of two arguments written in continuation-passing style:

$$\textbf{let } f(k, x) = (\textbf{let } g(j, y) = e \textbf{ in } k(g)) \textbf{ in } \ldots$$

The goal of the uncurrying transformation is to rewrite the body of g as a call to an uncurried helper function f':

$$\begin{aligned}&\textbf{let } f(k, x) = (\textbf{let } g(j, y) = f'(j, x, y) \textbf{ in } k(g))\\ &\textbf{and } f'(j, x, y) = e \textbf{ in } \ldots\end{aligned}$$

Though this doesn't uncurry f directly, we can record the variables f and g in a set to be returned along with the transformed expression, and tell a future inlining pass to inline fully saturated calls to those variables. This calls the uncurried f' where possible and keeps f available in case it's ever partially applied. If it turns out that f is never partially applied, then dead function elimination will (later) eliminate it.

In the general case (and ignoring some technical side-conditions), the uncurryer traverses lists of mutually recursive functions and replaces terms matching the pattern

$$(f(k :: \vec{x}) = \textbf{let } g(\vec{y}) = e \textbf{ in } k(g)) :: \vec{fd}$$

with

$$\begin{aligned}&(f(k :: \vec{x}) = \textbf{let } g(\vec{y}) = f'(\vec{x} +\!\!+ \vec{y}) \textbf{ in } k(g)) ::\\ &(f'(\vec{x} +\!\!+ \vec{y}) = e) :: \vec{fd}.\end{aligned}$$

Implementing this transformation as a recursive function using pattern matching is not difficult. If one wants to preserve globally unique bindings, the implementation must carry around a fresh-name generator (*i.e.,* a counter).

But one must choose carefully where to make recursive calls! A seemingly natural choice for a recursive call is the body e of the newly created uncurried helper function f', but such a choice would actually prevent any function with more than two arguments from being fully uncurried. To see why, consider the following example of a curried function f that takes four arguments x, y, z, and w:

$$\begin{aligned}&\textbf{let } f(k_f, x) =\\ &\quad \textbf{let } g(k_g, y) =\\ &\quad\quad \textbf{let } h(k_h, z) =\\ &\quad\quad\quad \textbf{let } j(k_j, w) = e_{\text{body}} \textbf{ in } k_h(j)\\ &\quad\quad \textbf{in } k_g(h)\\ &\quad \textbf{in } k_f(g)\\ &\textbf{in } \ldots\end{aligned}$$

This term matches the uncurrying pattern with

$$e := \textbf{let } h(k_h, z) = \cdots \textbf{ in } k_g(h),$$

and applying the rewrite rule yields the following transformed term:

$$\begin{aligned}&\textbf{let } f(k'_f, x') = (\textbf{let } g(k'_g, y') = f'(k'_f, x', k'_g, y') \textbf{ in } k'_f(g))\\ &\textbf{and } f'(k_f, x, k_g, y) =\\ &\quad \boxed{\begin{aligned}&\textbf{let } h(k_h, z) =\\ &\quad \textbf{let } j(k_j, w) = e_{\text{body}} \textbf{ in } k_h(j)\\ &\textbf{in } k_g(h)\end{aligned}}_{=e}\\ &\textbf{in } \ldots\end{aligned}$$

Immediately making a recursive call on e (boxed in red) would uncurry the inner h, after which the body of f' would no longer match the uncurrying pattern. This would prevent the 4-argument f that we started with from being fully uncurried.

The crux of the issue is that, in the list

$$\begin{aligned}&(f(k :: \vec{x}) = \textbf{let } g(\vec{y}) = f'(\vec{x} +\!\!+ \vec{y}) \textbf{ in } k(g)) ::\\ &(f'(\vec{x} +\!\!+ \vec{y}) = e) :: \vec{fd}\end{aligned}$$

reached after a single uncurrying step, one must first make a recursive call on $(f'(\vec{x} +\!\!+ \vec{y}) = e) :: \vec{fd}$ before transforming e, in case f' is itself eligible for uncurrying. Thus an efficient implementation must make sure to make a recursive call not on e, but on this entire sublist.

3 A THEORY OF EFFICIENT PROGRAM TRANSFORMATION

So far we have described several program transformations along with the ugly details required to get them right. Our tool allows to specify each of these transformations, complete with the various auxiliary data structures needed for efficiency in each case. Moreover, it derives large portions of efficient functional implementations from such specifications automatically.

To function, our tool requires formal descriptions of auxiliary data structures used by implementors. We codify such strategies in a "theory of efficient program transformation." The design of our theory is motivated by the following observations about the transformations described in the previous section:

- Many desirable transformations aren't local: for example, case- and projection-folding require knowledge of bindings in scope. Even uncurrying, which appears to be a simple local transformation, requires a mechanism for generating fresh names if one wants to maintain globally unique bindings.
- Each transformation can be implemented as a single top-down-then-bottom-up pass that runs in quasilinear time. To do so, one needs:
  - fine-grained control over which rules should be applied top-down vs. bottom-up and the locations of recursive calls,
  - extra parameters (e.g., bindings in scope) and state variables (e.g., occurrence count maps) that efficiently summarize information about the surrounding context, and
  - the ability to delay computations and fuse multiple such delayed computations together to avoid extra traversals (e.g., substitution).

From these observations, we assemble the following theory:

- A program transformation is specified by guarded contextual rewrite rules of the form $C[e] \longrightarrow C[e']$ if P, embellished with *labels* that control traversal order and the locations of recursive calls.
- A program transformation is implemented by a recursive function. We can think of this function as a traversal with zipper [Huet 1997]: at each recursive call on e, there exists some surrounding one-hole context C, initially empty, that grows as the function descends into e's subterms and shrinks as it returns from recursive calls. Viewed in this way, an execution trace of this recursive function corresponds to the execution of an abstract machine with states of the form $\langle C \mid e \rangle$. This abstract machine has transitions that correspond directly to the aforementioned guarded contextual rewrite rules, connecting the functional implementation to its specification.
- Auxiliary data structures manipulated by an efficient implementation summarize information about this machine state for easy lookup. For example, the map ρ of bindings in scope holds a mapping $x \mapsto c\ \vec{y}$ for each $\mathbf{let}\ x = \mathrm{Con}\ c\ \vec{y}\ \mathbf{in}\ \_$ that occurs in C. A data structure that only summarizes information about the surrounding context C can be *passed down* as an extra parameter; on the other hand, a data structure that summarizes information about both C and e (e.g., the map δ of global occurrence counts) must be *passed down and returned up* as a state variable.
- A computation can be efficiently delayed and fused so long as it distributes over constructor application. Though quite restrictive, we believe that this is still sufficient to express a wide range of program transformations.

3.1 Transformation as Traversal with Zipper

We now make the above high-level description more precise. Define the following syntactic categories:

$$
\begin{array}{llcl}
\text{Constructors} & \text{Constr} & \ni & c \\
\text{Atoms} & \text{Atom} & \ni & a \\
\text{Terms} & e & ::= & a \mid c\,\vec{e} \\
\text{Contexts} & C & ::= & \square \mid C :: c\,\vec{e}\,\square\,\vec{e}
\end{array}
$$

We consider terms e built out of atoms a and constructor applications $c\,\vec{e}$, and define one-hole contexts C to be snoc-lists of *frames* (contexts of depth 1) of the form $c\,\vec{e}\,\square\,\vec{e}$. We write $\square$ for the empty context and use square brackets to denote context application. The distinction between nullary constructors and atoms will be made clear when we describe delayed computations.

Recall that the recursive functions we would like to generate perform transformations in a top-down-then-bottom-up manner. What we mean by this is that our generated recursive functions should have two phases:

Top-down, some set of *top-down rewrite rules* are tried in turn and recursive calls are made; if no top-down rewrite rule is applicable, then recursive calls are simply made on every subterm. Atoms are left untouched. This phase inspects the input term e and produces a transformed term e'.

Bottom-up, after recursive calls have been made, some set of *bottom-up rewrite rules* are tried in turn; if no rules are applicable, then the term is left unchanged. This phase inspects the e' returned by the top-down phase and produces a final transformed term.

To enforce this traversal order, we define *labels*, labeled terms, and labeled contexts:

$$
\begin{array}{llcl}
\text{Labels} & \ell & ::= & \text{Down} \mid \text{Mid} \mid \text{Up} \mid \text{Shift} \\
\text{Labeled terms} & e^{\ell} & ::= & \ell\,e \mid c\,\vec{e}^{\ell} \\
\text{Labeled contexts} & C^{\ell} & ::= & \square \mid C^{\ell} :: \ell\,\square \mid C^{\ell} :: c\,\vec{e}\,\square\,\overrightarrow{e^{\ell}}
\end{array}
$$

In a labeled term, each subterm may be associated with a label ℓ; similarly a labeled context is composed of frames that may contain labeled terms and frames of the form $\ell\,\square$. Terms left of the hole $\square$ are unlabeled—our abstract machines (defined below) will process labelled subterms left-to-right, with $\square$ marking the boundary. Labels are used to control traversal order:

- Down e is a term e waiting to be traversed in a top-down manner.
- Mid e is a term e that has been traversed top-down and is waiting to be traversed bottom-up.
- Up e is a term that has been traversed both top-down and bottom-up, and is ready to be returned.
- Shift forces our abstract machines to move to the next argument when processing a constructor application $c\,\vec{e}$.

We define a program transformation in terms of an abstract machine operating on states $\langle C^{\ell} \mid e^{\ell} \rangle$ composed of a labeled one-hole context C^{ℓ} and focused term e^{ℓ}. A machine's behavior is defined by a list of transition rules $\langle C_1^{\ell} \mid e_1^{\ell} \rangle \longrightarrow \langle C_2^{\ell} \mid e_2^{\ell} \rangle$ if P, where C_1^{ℓ}, e_1^{ℓ}, C_2^{ℓ}, e_2^{ℓ}, and P may also contain variables denoting placeholders for concrete terms and contexts. The machine is executed by starting in some initial state $\langle C_0^{\ell} \mid e_0^{\ell} \rangle$ and repeatedly applying the first applicable rule in the list.

$$\vec{D}$$

$$\left\langle C^\ell \,\middle|\, \text{Down}\ (c\ \overrightarrow{e^\ell}) \right\rangle \longrightarrow \left\langle C^\ell :: \text{Mid}\ \square \,\middle|\, c\ (\overrightarrow{\text{Down}\ e^\ell}) \right\rangle \ (\textsc{congruence})$$

$$\left\langle C^\ell \,\middle|\, c\ e^\ell\ \overrightarrow{e^\ell} \right\rangle \longrightarrow \left\langle C^\ell :: c\ \square\ \overrightarrow{e^\ell} \,\middle|\, e^\ell \right\rangle \ (\textsc{first-argument})$$

$$\left\langle C^\ell :: c\ \vec{e}\ \square\ \overrightarrow{e^\ell} \,\middle|\, \text{Up}\ e \right\rangle \longrightarrow \left\langle C^\ell \,\middle|\, \text{Shift}\ (c\ \vec{e}\ e\ \overrightarrow{e^\ell}) \right\rangle \ (\textsc{exit-argument})$$

$$\left\langle C^\ell \,\middle|\, \text{Shift}\ (c\ \vec{e}\ e^\ell\ \overrightarrow{e^\ell}) \right\rangle \longrightarrow \left\langle C^\ell :: c\ \vec{e}\ \square\ \overrightarrow{e^\ell} \,\middle|\, e^\ell \right\rangle \ (\textsc{next-argument})$$

$$\left\langle C^\ell :: c\ \vec{e}\ \square \,\middle|\, \text{Up}\ e \right\rangle \longrightarrow \left\langle C^\ell \,\middle|\, \text{Up}\ (c\ \vec{e}\ e) \right\rangle \ (\textsc{last-argument})$$

$$\left\langle C^\ell \,\middle|\, c \right\rangle \longrightarrow \left\langle C^\ell \,\middle|\, \text{Up}\ c \right\rangle \ (\textsc{no-arguments})$$

$$\left\langle C^\ell \,\middle|\, a \right\rangle \longrightarrow \left\langle C^\ell \,\middle|\, \text{Up}\ a \right\rangle \ (\textsc{atom})$$

$$\left\langle C^\ell :: \text{Mid}\ \square \,\middle|\, \text{Up}\ e \right\rangle \longrightarrow \left\langle C^\ell \,\middle|\, \text{Mid}\ e \right\rangle \ (\textsc{exit-topdown})$$

$$\vec{U}$$

$$\left\langle C^\ell \,\middle|\, \text{Mid}\ e \right\rangle \longrightarrow \left\langle C^\ell \,\middle|\, \text{Up}\ e \right\rangle \ (\textsc{exit-bottomup})$$

Fig. 3. A program transformation $(\vec{D}, \vec{U})$ as an abstract machine.

We define a top-down rewrite rule to be a transition rule of the form

$$\left\langle C^\ell \,\middle|\, \text{Down}\ e_1 \right\rangle \longrightarrow \left\langle C^\ell :: \text{Mid}\ \square \,\middle|\, e_2^\ell \right\rangle \ \text{if}\ P$$

where e_2^ℓ can contain only Down labels. The Down label on the left-hand side forces these rules to be applicable only to terms waiting to be traversed top-down. The Down labels in e_2^ℓ are used to mark the subterms on which to make "recursive calls." The Mid label is used to indicate that bottom-up rewrite rules should be checked at this location after all such recursive calls have been completed. We explain exactly how each of these labels control traversal order in detail below. We also will require that Down labels in the right-hand side e_2^ℓ be applied only to variables. This restriction does not unduly limit expressiveness: x may be any arbitrary e because P could just include an extra clause of the form $x = e$. The restriction just makes the deriver easier to implement, and delayed computations a bit easier to explain.

We define a bottom-up rewrite rule to be a transition rule of the form

$$\left\langle C^\ell \,\middle|\, \text{Mid}\ e_1 \right\rangle \longrightarrow \left\langle C^\ell \,\middle|\, \text{Up}\ e_2 \right\rangle \ \text{if}\ P.$$

The Mid label on the left-hand side forces these rules to be applicable only to terms that have already been traversed top-down and are waiting to be traversed bottom-up. The Up label on the right-hand side marks the transformed term as having been traversed both top-down and bottom-up, and as ready to be returned.

A program transformation is a pair $(\vec{D}, \vec{U})$ of top-down rewrite rules $\vec{D}$ and bottom-up rewrite rules $\vec{U}$, and denotes the abstract machine in figure 3. Upon encountering an input term, this machine first tries to apply each of the top-down rewrite rules $\vec{D}$. If any such rule is applicable, the current subterm in focus is replaced by its right-hand side, which marks the locations of every

subsequent recursive call with a Down label and augments the context C^ℓ with the frame Mid $\square$ to indicate that bottom-up rewrite rules should be checked at this location after the top-down phase has completed. If no top-down rule is applicable and the machine is processing a constructor application $c\ \overrightarrow{e^\ell}$, the CONGRUENCE rule schedules each argument e^ℓ for a recursive call.

Rules FIRST-ARGUMENT, EXIT-ARGUMENT, NEXT-ARGUMENT, LAST-ARGUMENT, NO-ARGUMENTS, and ATOM carry out the rest of the top-down phase by processing subterms of the form e^ℓ containing Down labels indicating the locations of recursive calls. FIRST-ARGUMENT starts the process by descending into the first argument e^ℓ in the case where the focused term is a constructor application $c\ e^\ell\ \overrightarrow{e^\ell}$. After an argument e has been fully processed (is labeled Up), EXIT-ARGUMENT continues the computation by moving back upwards and using Shift to indicate that the next argument is ready to be processed. NEXT-ARGUMENT then descends into the next such argument e^ℓ. After the rightmost argument e has been processed, LAST-ARGUMENT collects it along with all previous arguments $\vec{e}$ and marks the whole constructor application $c\ \vec{e}\ e$ as done with the top-down phase and ready for the bottom-up phase. Finally, NO-ARGUMENTS and ATOM state that any nullary constructor application or atom is immediately done with the top-down phase.

After the top-down phase is complete, EXIT-TOPDOWN marks the beginning of the bottom-up phase by popping the Mid label from the context C^ℓ and applying the label to e. This allows the application of bottom-up rewrite rules $\vec{U}$. If no such rules are applicable, EXIT-BOTTOMUP marks e as done with the bottom-up phase.

3.1.1 A Concrete Example. Consider a constant folding transformation $0 \times e \longrightarrow 0$ over arithmetic expressions $e ::= 0 \mid 1 \mid e{+}e \mid e{\times}e$. An efficient implementation should perform this transformation in both a top-down and bottom-up manner:

- Rewriting top-down allows folding $0 \times e$ down to 0 without wastefully processing e.
- Rewriting bottom-up allows folding $(0 \times e_1) \times e_2$ down to 0 in one linear-time pass.

Such an implementation can be specified by two rules—one top-down and one bottom-up:

$$\vec{D} = \left[\left\langle C^\ell \,\middle|\, \text{Down}\ (0 \times e)\right\rangle \longrightarrow \left\langle C^\ell :: \text{Mid}\ \square \,\middle|\, 0\right\rangle\ \text{if}\ \top\right]$$
$$\vec{U} = \left[\left\langle C^\ell \,\middle|\, \text{Mid}\ (0 \times \text{e})\right\rangle \longrightarrow \left\langle C^\ell \,\middle|\, \text{Up}\ 0\right\rangle\ \text{if}\ \top\right]$$

The execution trace in figure 4 demonstrates how the corresponding abstract machine folds $(0 \times (1 + 1 + 1)) \times 1$ down to 0. Note that top-down rule application avoids processing the large expression $1 + 1 + 1$, saving work, and that the bottom-up rule application performs the reduction $0 \times 0 \longrightarrow 1$ after the inner $0 \times (1 + 1 + 1)$ has been reduced to 0, allowing all redexes to be eliminated in one pass.

3.1.2 Program Transformations as Recursive Functions. So far, we've shown how program transformations can be specified by lists of top-down and bottom-up rewrite rules, and executed by an abstract machine. We now connect the abstract machine to a functional implementation, by giving a translation from specifications $(\vec{D}, \vec{U})$ to recursive functions written in a generic untyped call-by-value functional language with pattern matching and pattern guards [Erwig and Jones 2001]. The translation is given in figure 5. Though we will improve upon this translation in later sections, the functions it produces already resemble the implementations generated by our tool.[2] The function $[\![-]\!]$ translates a specification of a program transformation $(\vec{D}, \vec{U})$ into a recursive function that implements it. It leaves behind a hole (boxed in red) for each rewrite rule in the specification. These holes are to be filled in manually by the user of our tool, with a pattern guard

[2]Coq does not support pattern guards, so in practice our tool emits case trees with join points resembling a naive desugaring of pattern guards into a standard ML-like language.

$\langle \square \mid \text{Down}\ (0 \times (1 + 1 + 1)) \times 1 \rangle \longrightarrow$ (CONGRUENCE)
$\langle \square :: \text{Mid}\ \square \mid \text{Down}\ (0 \times (1 + 1 + 1)) \times \text{Down}\ 1 \rangle \longrightarrow$ (FIRST-ARGUMENT)
$\langle \square :: \text{Mid}\ \square :: \square \times \text{Down}\ 1 \mid \text{Down}\ (0 \times (1 + 1 + 1)) \rangle \longrightarrow$ (top-down rule application)
$\langle \square :: \text{Mid}\ \square :: \square \times \text{Down}\ 1 :: \text{Mid}\ \square \mid 0 \rangle \longrightarrow$ (NO-ARGUMENTS)
$\langle \square :: \text{Mid}\ \square :: \square \times \text{Down}\ 1 :: \text{Mid}\ \square \mid \text{Up}\ 0 \rangle \longrightarrow$ (EXIT-TOPDOWN)
$\langle \square :: \text{Mid}\ \square :: \square \times \text{Down}\ 1 \mid \text{Mid}\ 0 \rangle \longrightarrow$ (EXIT-BOTTOMUP)
$\langle \square :: \text{Mid}\ \square :: \square \times \text{Down}\ 1 \mid \text{Up}\ 0 \rangle \longrightarrow$ (EXIT-ARGUMENT)
$\langle \square :: \text{Mid}\ \square \mid \text{Shift}\ (0 \times \text{Down}\ 1) \rangle \longrightarrow$ (NEXT-ARGUMENT)
$\langle \square :: \text{Mid}\ \square :: 0 \times \square \mid \text{Down}\ 0 \rangle \longrightarrow$ (CONGRUENCE)
$\langle \square :: \text{Mid}\ \square :: 0 \times \square :: \text{Mid}\ \square \mid 0 \rangle \longrightarrow$ (NO-ARGUMENTS)
$\langle \square :: \text{Mid}\ \square :: 0 \times \square :: \text{Mid}\ \square \mid \text{Up}\ 0 \rangle \longrightarrow$ (EXIT-TOPDOWN)
$\langle \square :: \text{Mid}\ \square :: 0 \times \square \mid \text{Mid}\ 0 \rangle \longrightarrow$ (EXIT-BOTTOMUP)
$\langle \square :: \text{Mid}\ \square :: 0 \times \square \mid \text{Up}\ 0 \rangle \longrightarrow$ (LAST-ARGUMENT)
$\langle \square :: \text{Mid}\ \square \mid \text{Up}\ (0 \times 0) \rangle \longrightarrow$ (EXIT-TOPDOWN)
$\langle \square \mid \text{Mid}\ (0 \times 0) \rangle \longrightarrow$ (bottom-up rule application)
$\langle \square \mid \text{Up}\ 0 \rangle$

Fig. 4. Running a simple constant folding transformation on the term $(0 \times (1 + 1 + 1)) \times 1$.

$$
\begin{aligned}
&\left[\!\left[(\vec{D}, \vec{U}) \right]\!\right] = \\
&\quad \textbf{let rec}\ \textit{transform}\ C\ e = \\
&\qquad \textbf{let}\ e' = \\
&\qquad\quad \textbf{if}\ e\ \textit{is an atom}\ \textbf{then}\ e\ \textbf{else} \\
&\qquad\quad \textbf{match}\ e\ \textbf{with} \\
&\qquad\qquad \textit{for each}\ \langle C \mid \text{Down}\ e_1 \rangle \longrightarrow \left\langle C :: \text{Mid}\ \square \,\middle|\, e_2^{\ell} \right\rangle\ \text{if}\ P \in \vec{D}, \\
&\qquad\qquad e_1 \mid \text{Some}\ (x, \ldots) \leftarrow \textit{guard implementing}\ P \Rightarrow \left[\!\left[e_2^{\ell} \right]\!\right]_C\ \textit{where}\ \text{FV}(P) = \{x, \ldots\} \\
&\qquad\qquad \textit{for each constructor}\ c, \\
&\qquad\qquad c\ \vec{e} \Rightarrow \left[\!\left[c\ \overrightarrow{(\text{Down}\ e)} \right]\!\right]_C \\
&\qquad \textbf{match}\ e'\ \textbf{with} \\
&\qquad\quad \textit{for each}\ \langle C \mid \text{Mid}\ e_1 \rangle \longrightarrow \langle C \mid \text{Up}\ e_2 \rangle\ \text{if}\ P \in \vec{U}, \\
&\qquad\quad e_1 \mid \text{Some}\ (x, \ldots) \leftarrow \textit{guard implementing}\ P \Rightarrow e_2\ \textit{where}\ \text{FV}(P) = \{x, \ldots\} \\
&\qquad\quad e \Rightarrow e
\end{aligned}
$$

Fig. 5. Generating a recursive function from a specification $(\vec{D}, \vec{U})$.

$$
\begin{aligned}
&[\![\text{Down } e]\!]_{C^\ell} = \textit{transform } C\ e \textit{ where } C = C^\ell \textit{ with all labels erased}\\
&[\![c\ e_1^\ell\ \cdots\ e_n^\ell]\!]_{C^\ell} =\\
&\quad \textbf{let } x_1 = [\![e_1^\ell]\!]_{C^\ell :: c\ \square\ e_2^\ell\ \cdots\ e_n^\ell} \textbf{ in}\\
&\quad \textbf{let } x_2 = [\![e_2^\ell]\!]_{C^\ell :: c\ x_1\ \square\ e_3^\ell\ \cdots\ e_n^\ell} \textbf{ in}\\
&\qquad\quad \vdots\\
&\quad \textbf{let } x_{n-1} = [\![e_{n-1}^\ell]\!]_{C^\ell :: c\ x_1\ \cdots\ x_{n-2}\ \square\ e_n^\ell} \textbf{ in}\\
&\quad \textbf{let } x_n = [\![e_n^\ell]\!]_{C^\ell :: c\ x_1\ \cdots\ x_{n-1}\ \square} \textbf{ in}\\
&\quad c\ x_1\ \cdots\ x_n
\end{aligned}
$$

Fig. 6. Helper function for the translation in figure 5.

that checks the side conditions of each rewrite rule. If a guard for side condition P evaluates to Some $(v, \ldots)$, then P must hold with instantiations $x \mapsto v$ for each x in the free variables of P that do not occur in the corresponding rule's left-hand side. In section 3.3.1, we will show an example of a program transformation with nontrivial holes of this form. In our Coq implementation, these holes correspond to proof obligations of type option $\Sigma_{x,\ldots} P$. This type ensures that they can only be filled with sound implementations.[3]

The translation $[\![-]\!]$ uses a helper function $[\![e^\ell]\!]_C$, defined in figure 6, to translate the right-hand sides e^ℓ of top-down rewrite rules. Specifically, the extra parameter C is used to compute the one-hole context to be passed to each recursive call (i.e., each subterm labelled Down in e^ℓ); this context must be updated throughout the top-down phase to reflect changes made by recursive calls on subterms.

Each piece of the translation for a transformation $(\vec{D}, \vec{U})$ corresponds directly to a piece of the abstract machine given in figure 3:

- The separation of the implementation into top-down and bottom-up phases corresponds to the rules EXIT-TOPDOWN and EXIT-BOTTOMUP and the use of labels Down, Mid, and Up.
- The if-then-else to check whether e is an atom corresponds to ATOM.
- The let-bindings generated by $[\![-]\!]_C$, which carefully update the one-hole context parameter with newly transformed subterms, correspond to applications of FIRST-ARGUMENT, EXIT-ARGUMENT, NEXT-ARGUMENT, LAST-ARGUMENT, and NO-ARGUMENTS.
- Each case arm of the form $C, c\ \vec{e} \Rightarrow \left[\!\!\left[c\ \overrightarrow{(\text{Down } e)}\right]\!\!\right]_C$ in the top-down phase corresponds to an application of the CONGRUENCE rule.
- All remaining case arms correspond to the rewrite rules from which they were generated.

3.1.3 Constant Folding as a Recursive Function. Continuing our constant folding example, figure 7 shows the recursive function generated by the above translation for the constant folding transformation described in section 3.1.1. The two holes guard_1 and guard_2, boxed in red, correspond to the side conditions of the two rewrite rules from which the recursive function was generated: guard_1 corresponds to the side condition of the top-down rewrite rule $\langle C^\ell \mid \text{Down } (0 \times e)\rangle \longrightarrow \langle C^\ell :: \text{Mid } \square \mid 0\rangle$ if $\top$, and guard_2 corresponds to the side condition of the bottom-up rewrite rule

[3] Actually, to avoid the extra overhead associated with bundling everything up into option and sigma types, the proof obligation is written in continuation-passing style, with type $\Pi_R(\Pi_{x,\ldots} P \to R) \to R \to R$.

```
let rec transform C e =
  let e' =
    if false then e else
    match e with
      0 × e | Some () ← guard_1 ⇒ 0
      e_1 + e_2 ⇒
        let x_1 = transform (C :: □ + e_2) e_1 in
        let x_2 = transform (C :: x_1 + □) e_2 in
        x_1 + x_2
      e_1 × e_2 ⇒
        let x_1 = transform (C :: □ + e_2) e_1 in
        let x_2 = transform (C :: x_1 + □) e_2 in
        x_1 × x_2
  match e' with
    0 × e | Some () ← guard_2 ⇒ e
    _, e ⇒ e
```

Fig. 7. Constant folding, generated from the specification in section 3.1.1 according to the translation in figure 5.

$\langle C^\ell \mid \text{Mid } (0 \times \text{e}) \rangle \longrightarrow \langle C^\ell \mid \text{Up } 0 \rangle$ if $\top$. In this toy example, both side conditions are just the trivial side condition $\top$, and contain no free variables; therefore, both holes are trivial to fill with the always-successful guard.

Note that a majority of the implementation shown in figure 7 has to do with making recursive calls in unsurprising places, and is generated fully automatically by the translation. This toy example only has two such congruence cases, but in real applications there are many more, and our tool generates correctness proofs for each such case in addition to their implementations.

3.2 Summarizing Information in the Surrounding Context

We've shown that program transformations can be specified by contextual rewrite rules and executed on an abstract machine whose behavior corresponds closely to the execution of actual implementations written using recursion and pattern matching. We now use this correspondence to formally describe the kinds of auxiliary data structures frequently used by compiler writers to summarize information about the surrounding context.

Throughout this section, let C denote the context constructed by erasing all labels from C^ℓ, and similarly let e denote the term constructed by erasing all labels from e^ℓ. An auxiliary data structure is an extra value v that's related to the erased machine state $\langle C \mid e \rangle$ by some *invariant* $\sim$ at each execution step. This extra value evolves in parallel with the machine state; i.e., for each execution trace $\langle C_1^\ell \mid e_1^\ell \rangle \longrightarrow \cdots \longrightarrow \langle C_n^\ell \mid e_n^\ell \rangle$, there is a parallel trace $v_1, \ldots, v_n$ such that $\langle C_i \mid e_i \rangle \sim v_i$ for all $1 \leq i \leq n$.

Here are a few example specifications of such auxiliary data structures, drawn from the transformations described in section 2:

- The variable binding map ρ used by case (§2.1) and projection folding (§2.2) has invariant

$$\langle C \mid \_\rangle \sim \rho = \forall (x \mapsto c\ \vec{y}) \in \rho,\ C \text{ contains the binding } (\textbf{let } x = \text{Con } c\ \vec{y}\ \textbf{in } \_).$$

- The occurrence count map δ used by dead function elimination (§2.3) has invariant

$$\langle C \mid e\rangle \sim \delta = \forall x,\ \delta(x) = \text{no. of times } x \text{ occurs in non-binding position in } C[e].$$

- The monotonic counter variable x used by uncurrying (§2.4) to generate globally fresh names has invariant

$$\langle C \mid e\rangle \sim x = \forall y,\ y \text{ occurs in } C[e] \implies y < x.$$

Note that ρ's invariant doesn't depend on the subterm in focus e, while δ's and x's do. This has an interesting consequence: if $\langle C \mid c\ e\rangle \sim \rho$ and the recursive traversal of e produces a transformed term e', we still have $\langle C \mid c\ e'\rangle \sim \rho$ because ρ's invariant depends only on C. Thus, there is no need to update ρ "on the way up" after returning from a recursive traversal of a subterm. This is what allows ρ to be passed around as a parameter, while δ and x must be passed around as state variables.

So we have identified two kinds of auxiliary data structures used to summarize information about the surrounding context—parameters and state variables—and a way to formally specify their behavior using invariants. Our tool uses this formal description to generate, from a small set of user-defined basic operations, proved-correct code that performs the necessary bookkeeping to maintain each auxiliary data structure's invariants throughout a whole traversal. This is made possible by the observation that, in order to preserve an invariant $\sim$ throughout a whole traversal, one need only manually specify how to preserve it across each abstract machine step. Inspecting the transition rules in figure 3 reveals that there are only a few such operations that need to be implemented manually:

- The rules CONGRUENCE, ATOM, EXIT-TOPDOWN, and EXIT-BOTTOMUP all erase to the identity transition $\langle C \mid e\rangle \longrightarrow \langle C \mid e\rangle$, and thus it's trivial to generate code that preserves invariants across these transitions.
- Top-down and bottom-up rewrite rules both erase to transitions of the form

$$\langle C \mid e_1\rangle \longrightarrow \langle C \mid e_2\rangle \text{ if } P,$$

  and one must manually specify how to preserve invariants on state variables across this transition. (It's not necessary to specify how to preserve invariants on parameters because the surrounding context C remains unchanged.) This corresponds to the fact that the user must explain how to preserve the invariants on their auxiliary data structures across individual rewriting steps.
- The rules FIRST-ARGUMENT, EXIT-ARGUMENT, NEXT-ARGUMENT, and LAST-ARGUMENT erase to transitions of the form

$$\langle C \mid c\ \vec{e}_1\ e\ \vec{e}_2\rangle \longrightarrow \langle C :: c\ \vec{e}_1\ \square\ \vec{e}_2 \mid e\rangle \text{ and}$$
$$\langle C :: c\ \vec{e}_1\ \square\ \vec{e}_2 \mid e\rangle \longrightarrow \langle C \mid c\ \vec{e}_1\ e\ \vec{e}_2\rangle\,.$$

  Since both the context and focus are changing in these transitions, both state variables and parameters need to be updated. These transitions correspond to the fact that the user must explain how to preserve the invariants on their auxiliary data structures across single downward or upward movements through the term being transformed.

3.3 Delayed Computation

Delayed computations are used to incrementalize and fuse together computations that would otherwise require extra traversals. In the examples presented in section 2, this was used to efficiently implement variable substitution: rather than calling a helper function for substituting one variable by another, projection folding uses a delayed substitution σ that gets applied to free variables throughout the traversal.

While parameters and state variables are best characterized by viewing program transformations as abstract machines, delayed computations are best characterized by viewing program transformations as recursive functions. Suppose *transform* is a recursive function that implements a program transformation $(\vec{D}, \vec{U})$ without using a delayed computation, as shown in figure 5. For simplicity's sake, we'll also assume that *transform* doesn't use a state variable or parameter—the explanation to follow can be readily generalized to account for them, but involves a number of extra technical details that obscure the high-level idea. *transform*'s body has the following form:

```
let rec transform C e =
  let e' =
    if e is an atom then e else
    match e with
      ... top-down rules ...
      ... congruence cases ...
  match e' with
    ... bottom-up rules ...
```

We want a function *transform′* that takes an extra parameter d representing a delayed computation and satisfies $\mathit{transform}'\ C\ e\ d = \mathit{transform}\ C\ (\mathit{run}\ d\ e)$, where *run* is a function that takes the delayed computation d and runs it to completion on e. (For example, in the case of projection folding, d is a partial map $\mathrm{Var} \rightharpoonup \mathrm{Var}$ and *run* a function that implements parallel substitution.) Importantly, we would like to implement *transform′* but without doing all of $\mathit{run}\ d$ before executing *transform*, and we would like to fuse nested calls $\mathit{run}\ d_1\ (\mathit{run}\ d_2\ e)$ into single calls $\mathit{run}\ d_3\ e$.

We now demonstrate how to derive such a *transform′* given suitable d. Along the way, we will clarify what properties d and *run* need to satisfy in order for this derivation to be possible. First, *transform′* must be initializable with some d that represents an identity computation, since there is no delayed computation pending at the very start of a transformation. That is, there must be some d_{id} such that $\mathit{run}\ d_{\mathrm{id}} = \mathit{id}$.

Now, let's take the equation $\mathit{transform}'\ C\ e\ d = \mathit{transform}\ C\ (\mathit{run}\ d\ e)$ as a naive definition for *transform′* and inline *transform* inside of it. This yields:

```
let rec transform' C e d =
  let e' =
    if (run d e) is an atom then run d e else
    match run d e with
      ... top-down rules ...
      ... congruence cases ...
  match e' with
    ... bottom-up rules ...
```

We want to determine whether *run d e* is an atom, and, if not, which top-down match clause to take, all without actually running (*run d*) to completion. At the same time, we still want the match expression to be compiled to a efficient series of conditional branches. To allow this, we require that (*run d e*) be an atom iff *e* is an atom, and that (*run d*) distribute over constructor applications. Then any pattern *p* matches (*run d e*) if *p* matches *e*, and we can push the calls to (*run d*) inside the if-then-else and the pattern match:

```
let rec transform′ C e d =
  let e′ =
    if e is an atom then run d e else
    match e with
      ... top-down rules ...
      ... congruence cases ...
  match e′ with
    ... bottom-up rules ...
```

Now, if *transform′* is to satisfy *transform′ C e d* = *transform C* (*run d e*), we need to adjust each top-down match clause: since we have removed the call to (*run d*), the pattern variables in each branch capture subterms of *e* as they are *before* running *d*, not after. We must modify the right-hand side of each match clause to perform a bit of the delayed computation alongside whatever rewrite rule it implements. There are two kinds of match clauses in this match expression: (1) clauses that correspond to top-down rewrite rules, and (2) clauses for congruence cases. We describe only how to modify (1). The same method can be used to modify (2) because each congruence case for $c\ \vec{e}$ is equivalent to a top-down rewrite rule $\langle C^\ell \mid \text{Down}\ (c\ \vec{e})\rangle \longrightarrow \left\langle C^\ell :: \text{Mid}\ \square \;\middle|\; c\ \overrightarrow{(\text{Down}\ e)}\right\rangle$ if $\top$.

Suppose we're given a clause corresponding to a rule $\langle C^\ell \mid \text{Down}\ e_1\rangle \longrightarrow \langle C^\ell :: \text{Mid}\ \square \mid e_2^\ell\rangle$ if P. We modify it as follows: because (*run d*) distributes over constructor applications, we can repeatedly apply distributivity to find a delayed subcomputation d_x for each variable x in the pattern e_1, such that $run\ d\ e_1 = e_1[\overrightarrow{run\ d_x\ x/x}]$. We use $\vec{x}$ to refer to a list of these variables, and $\vec{d}_x$ to refer to the corresponding list of delayed computations. Recall that our generator requires users to manually write a guard to check the side condition P. Given the context C, pattern variables $\vec{x}$, and corresponding subcomputations $\vec{d}_x$, this guard must either fail by returning None or succeed with Some $(\ldots)$ containing instantiations for each free variable in P that doesn't occur in the left-hand side e_1. To properly incrementalize (*run d*), we update this guard's specification as follows:

- Divide the free variables of P into two parts: let $\vec{y}$ be the variables where Down y appears in the right-hand side e_2^ℓ (i.e., the subterms on which to recur), and $\vec{z}$ all other variables that do not occur in the left-hand side. We allow each y in $\vec{y}$ to have a corresponding delayed computation d_y. Then, where *transform* makes a recursive call on *run d y*, *transform′* makes a recursive call on y with extra argument d_y. This change is precisely what allows for the fusion of nested delayed computations; we will illustrate it in action below with a concrete example.
- Because $\vec{x}$ represents pre-delayed-computation values, the guard should succeed only if P holds on *post-delayed-computation* values $\overrightarrow{run\ d_x\ x}$, *not* on the captured pattern variables $\vec{x}$.
- The guard's specification then becomes the following: if running the guard given C, $\vec{x}$, and $\vec{d}_x$ yields Some $(\vec{y}, \vec{d}_y, \vec{z})$, then $P[\overrightarrow{run\ d_x\ x/x}][\overrightarrow{run\ d_y\ y/y}]$; i.e., the side condition P must hold under the substitutions of each x by $(run\ d_x\ x)$ and y by $(run\ d_y\ y)$.

This is all quite tricky to work out by hand, and our example below will show the kinds of bugs that can easily arise when manipulating delayed computations manually. Fortunately, our tool prevents the user from going wrong: for each hole to be filled with a side condition check, we generate the above specification in the form of a Coq proof obligation, ensuring that the user can't fill the hole with incorrect code.

Finally, like parameters and state variables, delayed computations may have an invariant; this invariant may depend on the focused term, and must be preserved at every recursive call.

3.3.1 A Concrete Example of Delayed Computation. Recall that the goal of the projection folding transformation is to statically evaluate projections, so that

$$\ldots\ \mathbf{let}\ x = (y, z)\ \mathbf{in}\ \ldots\ \mathbf{let}\ w = \mathrm{fst}\ x\ \mathbf{in}\ e$$

is optimized to

$$\ldots\ \mathbf{let}\ x = (y, z)\ \mathbf{in}\ \ldots\ e[y/w].$$

In general, if we ever encounter a term $\mathbf{let}\ z = \mathrm{Proj}_n\ x\ \mathbf{in}\ e$ in a context C that contains a binding $\mathbf{let}\ x = \mathrm{Con}\ c\ \vec{y}\ \mathbf{in}\ \_$, we would like to rewrite $\mathbf{let}\ z = \mathrm{Proj}_n\ x\ \mathbf{in}\ e$ into $e[y_n/z]$, where y_n denotes the nth component of $\vec{y}$. Assuming that all variable bindings are globally unique, we can express this as the following top-down rewrite rule (for clarity, we have written the side condition above a horizontal line, as in inference rule notation):

$$\frac{C = D \circ (\mathbf{let}\ x = \mathrm{Con}\ c\ \vec{y}\ \mathbf{in}\ \square) \circ E \qquad e' = e[y_n/z]}{\langle C \mid \mathrm{Down}\ (\mathbf{let}\ z = \mathrm{Proj}_n\ x\ \mathbf{in}\ e)\rangle \longrightarrow \langle C :: \mathrm{Mid}\ \square \mid \mathrm{Down}\ e'\rangle}$$

As explained in section 2.2, an efficient implementation of this rule delays the substitution $[y_n/z]$, to be combined later with other substitutions into a big parallel substitution σ. In terms of the d and *run* function discussed in the previous section, our d in this case is just σ and *run* a function *subst* that performs the parallel substitution.

Now, in the recursive function our tool generates from this specification, there will be a match clause that corresponds to the above rewrite rule. In this match clause, there is some delayed substitution σ, the term being transformed has matched against the pattern $\mathbf{let}\ z = \mathrm{Proj}_n\ x\ \mathbf{in}\ e$, and the user of our tool must manually write a guard that checks that the rule's side condition holds of $\mathit{subst}\ \sigma\ (\mathbf{let}\ z = \mathrm{Proj}_n\ x\ \mathbf{in}\ e)$. By distributivity, it's sufficient to check that the condition holds of $\mathbf{let}\ (\sigma z) = \mathrm{Proj}_n\ (\sigma x)\ \mathbf{in}\ (\mathit{subst}\ \sigma\ e)$.[4]

Recall the method described in the previous section for generating this guard's specification:

- Divide the free variables of the side condition P to be implemented into the variables $\vec{y}$ marked with a Down label in the right-hand side of the rule (i.e., the subterms on which to recur), and all other variables that don't occur in the left-hand side of the rule $\vec{z}$.
- Given the surrounding context C, pre-delayed-computation variables $\vec{x}$ and corresponding subcomputations $\vec{d_x}$, the guard must either fail with None or succeed with Some $(\vec{y}, \vec{d_y}, \vec{z})$ such that $P[\overrightarrow{run\ d_x\ x/x}][\overrightarrow{run\ d_y\ y/y}]$.

In this example, the captured variables z, x, n, e have corresponding delayed subcomputations σ, σ, $\emptyset$ (the identity substitution), and σ; that is, $\vec{x} = [z, x, n, e]$ and $\vec{d_x} = [\sigma, \sigma, \emptyset, \sigma]$. The free variables of the side condition to implement are $\{C, D, x, c, \vec{y}, E, e', e, z\}$; of these, only e' is marked with a Down label on the right-hand side Down e', and only D, c, $\vec{y}$, and E do not appear in the left-hand side $\mathbf{let}\ z = \mathrm{Proj}_x\ n\ \mathbf{in}\ e$. So, in this example, $\vec{y} = [e']$ and $\vec{z} = [D, c, \vec{y}, E]$.

[4]Currently, our tool does not perform this rewrite automatically; instead, it assumes that the user has chosen a properly distributive delayed computation and will be able to perform this step by hand. In future versions of our tool, we hope to automate this process.

Therefore, this particular guard has the following specification: given a context C, captured variables z, x, n, and e, and their respective delayed computations σ, σ, $\emptyset$, and σ, the guard the user implements must either fail with None or succeed with Some $(e', \sigma_{e'}, D, c, \vec{y}, E)$ such that

$$C = D \circ (\textbf{let } (\sigma x) = \text{Con } c\ \vec{y} \textbf{ in } \square) \circ E$$
$$\textit{subst } \sigma_{e'}\ e' = (\textit{subst } \sigma\ e)[y_n/z].$$

Such a specification could be implemented as follows:

$$\begin{aligned}
&\textbf{let} \text{ guard } C\ z\ x\ n\ e\ \sigma = \\
&\quad \textbf{if } C \textit{ can be written as } (D \circ \textbf{let } (\sigma x) = \text{Con } c\ \vec{y} \textbf{ in } \square \circ E) \textit{ for some } D, c, \vec{y}, E \\
&\quad \textbf{then} \text{ Some } (e, \sigma[z \mapsto y_n], D, c, \vec{y}, E) \\
&\quad \textbf{else} \text{ None}
\end{aligned}$$

where the check to see whether C can be written as a composition of contexts containing D, c, $\vec{y}$, and E can be implemented as a recursive function that searches through C for bindings of σx.

This specification gives the user an opportunity to come up with a new delayed substitution $\sigma_{e'}$ that satisfies the desired fusion law, which will be passed as argument to a recursive call on an e' of their choice. In the above implementation, we chose $e' := e$ and $\sigma_{e'} := \sigma[z \mapsto y_n]$, which correctly fuses the substitutions σ and $[y_n/z]$ and then makes a recursive call on e.[5]

Note also that the specification generated by the procedure described above defends the user against easy-to-make mistakes: it requires that the user search for σx in the context C, not x, and that they update the substitution σ with $z \mapsto y_n$, not $z \mapsto \sigma y_n$. Both of these choices, which take at least a moment of careful thought to make correctly when implementing projection folding by hand, are made mechanically by our tool's procedure for deriving a guard's specification, eliminating possibility for error.

Of course, as described in section 3.2, the point of an efficient rewriter is that it doesn't have to do things like search through an entire context C looking for bindings of σx, and an efficient implementation would instead maintain an auxiliary data structure ρ mapping variable names to their definitions for easy lookup. Though the above explanation of delayed computation has been simplified to more clearly explain how we generate each guard's specification in a way that allows nested delayed computations to fuse, our tool can just as well generate implementations that make use of delayed computations alongside the parameters and state variables discussed in section 3.2.

4 IMPLEMENTATION

At a high level, our tool is intended to be used as follows:

(1) The user writes down a first-order Coq inductive type representing the syntax of terms that they would like their program transformations to operate on (e.g., for the examples described in section 2, the user would write down the exp type shown in figure 2).
(2) The user runs a MetaCoq [Sozeau et al. 2020] program (which we have written) to generate, from this inductive type, a corresponding type of one-hole contexts for use in writing down contextual rewrite rules.
(3) The user uses these two inductive types to write down a list of top-down and bottom-up rewrite rules, expressed as a Coq inductive relation.
(4) The user specifies what kind of parameters and state variables are needed for the implementation by writing down their respective invariants, and then fills out typeclass instances

[5]This fusion is only sound under the assumptions that (1) σ satisfies the invariant that σ's domain and range are disjoint from e's bound variables and (2) the term being transformed has globally unique bindings. Though we gloss over these details in our explanation, our tool can in fact encode both of these invariants.

that explain how such invariants can be preserved across single upwards and downwards movements through the term being transformed.

(5) The user specifies what kind of delayed computation is needed by writing down a suitable *run* function and identity computation d_{id}.

(6) The user initiates a proof in Coq with a goal that mentions each of the above pieces of their specification, and runs our tool by applying a special tactic that we have written. This tactic, using a mix of MetaCoq and Ltac, parses the given specification from the proof context and synthesizes most of a functional implementation along with a proof that its inputs and outputs are related by the reflexive transitive closure of the user-supplied rewriting relation. For each rewrite rule R in the given specification, our tactic leaves behind two holes for the user to fill in: one corresponds to the guard responsible for checking R's side condition, and the other requires the user to explain how to preserve their parameters' and state variables' invariants across an application of R. For each congruence case corresponding to a constructor c, our tactic leaves behind a single hole that requires the user to prove that their delayed computation distributes over c. Each hole's type includes proof obligations for the correctness of its implementation, so that any well-typed filling of these holes will lead to a sound implementation.

We now describe each piece of the implementation in detail.

4.1 Generating a Type of One-Hole Contexts

In section 3.1, we defined one-hole contexts as snoc-lists of frames (i.e., contexts of depth 1). This definition gets slightly more complicated in a typed setting. Suppose we would like our program transformations to operate on the exps representing our λ_{cps} intermediate language, defined in figure 2. Then, frames can be built not only out of applications of exp constructors, but also out of constructors of any inductive types used in its definition: for example, there must be frames for applications of the Fcons constructor for constructing lists of mutually recursive function bodies. Thus, our Coq encoding of frames must be able to represent constructors of multiple different types. We accomplish this using the following Frame typeclass, which represents a finite universe U of simple inductive types and a type of frames for that universe:

```
Class Frame (U: Set) := {
  univD: U → Set;
  frame_t: U → U → Set;
  frameD: ∀ {A B: U}, frame_t A B → univD A → univD B }.
```

Here, U is an inductive type representing all inductive types used in the definition of a given root type (exp in our case), univD maps each $u : U$ to the type it represents, frame_t is a type constructor for frames in U, with each frame indexed by representatives of its hole and root type, and frameD represents application of frames. We then define one-hole contexts as a stack of frames where the hole and root types of adjacent frames agree:

```
Inductive frames_t' {U: Set} {F: U → U → Set}: U → U → Set :=
| frames_nil: ∀ {A}, frames_t' A A
| frames_cons: ∀ {A B C}, F A B → frames_t' B C → frames_t' A C.
Definition frames_t {U: Set} {Frame U} := frames_t' frame_t.
```

We can then define application and composition of one-hole contexts for all instances of the Frame typeclass at once: context application C[e] simply applies the frames in C to e one by one using frameD, and context composition C >++ D concatenates the two lists C and D together.

Writing an instance of the Frame typeclass by hand would be very tedious. Therefore, we use MetaCoq [Sozeau et al. 2020] to generate these instances automatically. For the exp type, the following Coq command runs the MetaCoq program mk_Frame_ops (which we have written) to generate a proper Frame instance named exp_Frame_ops for our exp type:

```
MetaCoq Run (mk_Frame_ops "exp" exp   [var; fun_tag; ctor_tag; prim; N]).
```

The first argument to mk_Frame_ops is a string to be prefixed to the objects it defines (in this case, exp_Frame_ops). The second argument is the type that we'd like our program transformations to operate on (which we will refer to as the *root type*), and the third argument is the set of types that we would like to be considered atoms. The MetaCoq program also generates a small database of miscellaneous information useful to our tool called exp_aux_data. After having run the above command, users must register both exp_Frame_ops and exp_aux_data as instances of the Frame and AuxData typeclasses respectively:

```
Instance AuxData_exp: AuxData exp_univ := exp_aux_data.
Instance Frame_exp: Frame exp_univ := exp_Frame_ops.
```

4.2 Specifying Rewrite Rules

With a suitable type of one-hole contexts in hand, a user can write down a list of top-down and bottom-up rewrite rules as a Coq inductive relation $(\rightsquigarrow) : \mathsf{Root} \to \mathsf{Root} \to \mathsf{Prop}$, where Root is the root type; each constructor in $(\rightsquigarrow)$'s definition corresponds to a single rewrite rule. A top-down rewrite rule $\langle C \mid \mathrm{Down}\ e_1 \rangle \longrightarrow \langle C :: \mathrm{Mid}\ \square \mid e_2 \rangle$ if P is written as a constructor with type

$$\forall \ldots \textit{variables used in the rule} \ldots, P \to C[e_1] \rightsquigarrow C[e_2\ \textit{with}\ \mathrm{Down}\ \textit{replaced by}\ \mathrm{Rec}]$$

For example, the projection folding rewrite rule shown in section 3.3.1 can be written as the following constructor:

```
| pfold_rule : ∀ C D E x c ys y y' i e,
  C = D >++ Econstr x c ys frames_nil >++ E ∧
  nth_error ys i = Some y' ∧
  e' = subst y' y e →
  C[Eproj y x i e] ⇝ C[Rec e'].
```

where we assume subst is a user-written function, defined elsewhere, that performs substitution of a single variable. The Rec operator, just like the Down label, is used to indicate the locations of recursive calls in top-down rewrite rules; it's just given a different name in our Coq implementation to make its meaning more apparent. A bottom-up rewrite rule $\left\langle C^{\ell} \,\middle|\, \mathrm{Mid}\ e_1 \right\rangle \longrightarrow \left\langle C^{\ell} \,\middle|\, \mathrm{Up}\ e_2 \right\rangle$ if P is written as a constructor with type

$$\forall \ldots \textit{variables used in the rule} \ldots, P \to \mathsf{BottomUp}\ (C[e_1] \rightsquigarrow C[e_2])$$

The symbols Rec and BottomUp are only used by our tool for specifying whether a rule should be considered top-down or bottom-up, and for marking the locations of recursive calls. Both symbols are defined as the identity function, so these specifications are definitionally equal to the standard Coq presentations of such rewrite rules. This is useful when the user wishes (outside of our tool) to prove, for example, that the rules preserve some evaluation relation for their language.

4.3 Specifying Parameters and State Variables

Parameters and state variables are completely specified by their invariants. Let Root be the root type and assume there is a Frame U instance for some universe U. A parameter is specified by a type R and an invariant that relates values of type R to one-hole contexts:

```
I_R : ∀ (A : U), frames_t A Root → R → Prop.
```

A state variable is specified by a type S and an invariant that relates values of type S to contexts and focused terms:

```
I_S : ∀ (A : U), frames_t A Root → univD A → S → Prop.
```

From this we define the types Param of parameters and State of state variables as sigma types with respect to these invariants:

```
Definition Param {A} C : Set := {r | I_R C r}.
Definition State {A} C e : Set := {s | I_S C e s}.
```

Recall that, in order for our tool to generate code that preserves these invariants throughout a traversal, the user must explain how to preserve them across single upwards and downwards movements. These obligations are encoded by the following typeclasses (>:: is frame-snoc):

```
Class Preserves_S_up := preserve_S_up : ∀ A B fs f x,
  State (fs >:: f) x → State fs (frameD f x).
Class Preserves_R := preserve_R : ∀ A B fs f,
  Param fs → Param (fs >:: f).
Class Preserves_S_dn := preserve_S_dn : ∀ A B fs f x,
  State fs (frameD f x) → State (fs >:: f) x.
```

4.4 Specifying Delayed Computations

Like parameters and state variables, a delayed computation is specified by a type D and an invariant that are then packaged into a sigma type:

```
Context (I_D : ∀ (A : U), univD A → D → Prop).
Definition Delay {A} e : Set := {d | I_D e d}.
```

The user must also provide an instance of the Delayed typeclass:

```
Class Delayed := {
  delayD : ∀ {A} (e : univD A), Delay e → univD A;
  delay_id : ∀ {A} (e : univD A), Delay e;
  delay_id_law : ∀ {A} (e : univD A), delayD (delay_id e) = e }.
```

The delayD method corresponds to the *run* function from section 3.3 and (delay_id, delay_id_law) is the identity delay d_{id} and its correctness proof.

4.5 Making Rewriters from Specifications

Given a root type Root, inductive relation Rstep, parameter, state variable, and delayed computation, a program transformation along with its correctness proof can be expressed by a dependently typed function

```
transform : Fuel → ∀ A (C : frames_t A Root) (e : univD A) (d : D e),
Param C → State C (delayD e d) → result C (delayD e d)
```

where Fuel is a fuel-parameter used to convince Coq that the recursive function generated by our tool terminates.

A result is a dependent record type that carries an updated state variable and a proof that the transformed expression is related to the original input expression (delayD e d) by the reflexive transitive closure of the rewriting relation Rstep:

```
Record result A C e : Set := mk_result {
  resTree : univD A;
  resState : State C resTree;
  resProof : clos_refl_trans Rstep (C [ e ]) (C [ resTree ]) }.
```

Setting the type of transform above as a proof goal and running the tactic mk_rw (which we have written) generates most of a functional implementation and leaves behind proof obligations to be filled in manually.

4.6 Some Important Optimizations

We have slightly simplified the presentation of our Coq implementation above. As described, it has a few undesirable inefficiencies: the generated function uses a fuel-parameter in cases where it may not be necessary, and carries around an extra context parameter that is relevant only to the correctness proof. In our actual implementation, we allow the user to optionally specify a termination metric, in which case our tool omits the fuel parameter and generates proof obligations that require the user to prove the metric decreases at every recursive call. We also use a lightweight technique making use of Coq's Prop erasure to ensure that the context parameter does not appear in the extracted OCaml despite still being usable in proofs.

5 EVALUATION

We used our tool to implement uncurrying as described in section 2.4, an inlining pass parameterized by an arbitrary inlining heuristic, and a *partial shrink reduction* pass that simultaneously performs case folding, projection folding, and dead variable elimination (but not inlining of functions called only once or dead function elimination). Each pass was implemented as part of the backend of the CertiCoq compiler [Anand et al. 2017].

Our implementations of uncurrying and partial shrinking require significantly less Coq code to write and prove correct. Our uncurryer is implemented and proved correct with respect to a system of rewrite rules in 329 lines. The manually written uncurryer requires 309 lines for the implementation alone; its specification and proof requires an additional 1323 lines. Our tool-based partial shrinker is implemented and proved correct in 2132 lines. The manually written full shrink reduction algorithm [Savary Bélanger and Appel 2017] is implemented in 1414 lines and proved correct with respect to a system of shrink rewrite rules in 9828 lines, giving 11252 in total. While our partial shrinker does not perform shrink inlining (inlining of functions called only once) or dead function elimination, which require an extra helper data structure with a tricky invariant, we believe that it would be possible to extend our implementation to support these additional optimizations while staying well under this line count.

In contrast, our tool-derived inliner is not significantly smaller than the manually written inliner. Our tool-derived inliner is implemented and proved correct in 1945 lines, of which 1300 implement and prove correct an α-renaming function. This function is not needed by our hand-written algorithm, which does substitution in a different way.[6] The implementation and proof for

[6]Our tool-derived inliner calls an α-renaming function on function bodies before inlining them. This incurs an extra traversal, but does not change the inliner's asymptotic complexity; it also ensures that, if there isn't much inlining to do in the input term, the names of most bound variables will be preserved. In contrast, our hand-written algorithm simply renames every bound variable it encounters, using an extra parameter to keep track of the current renaming as it descends into subterms and a counter variable to generate fresh names.

Table 1. Duration of the first (partial) shrink reduction pass when compiling a suite of benchmarks, averaged across 10 runs.

Benchmark	Hand-written	Tool-derived
vs_easy	36.41 ± 1.41 ms	50.43 ± 2.60 ms
vs_hard	39.45 ± 0.78	46.81 ± 0.87
binom	18.19 ± 0.27	10.16 ± 0.48
color	138.18 ± 2.04	132.68 ± 1.20
sha	49.93 ± 4.37	64.88 ± 1.37

Table 2. Duration of the uncurrying pass when compiling a suite of benchmarks, averaged across 10 runs.

Benchmark	Hand-written	Tool-derived
vs_easy	5.71 ± 0.33 ms	4.61 ± 0.06 ms
vs_hard	9.24 ± 0.37	3.92 ± 0.13
binom	0.98 ± 0.01	0.79 ± 0.01
color	6.63 ± 0.09	6.73 ± 0.10
sha	3.90 ± 0.13	3.84 ± 0.06

the manually written inliner totals 2966 lines. However, this comparison is somewhat apples-to-oranges: the hand-written inliner additionally has (limited) support for inlining of nontail calls as part of an ANF backend while our generated inliner is restricted to λ_{cps} terms, and its proof establishes the much stronger property that inputs and outputs are contextually equivalent.

We assess the efficiency of our generated implementations by comparing run times to those of their hand-written counterparts. CertiCoq's backend is a composition of multiple small passes, including uncurrying, interleaved between calls to the shrink reducer. We measure the duration of the uncurrying pass and the first (partial) shrinking pass[7] when compiling CertiCoq's benchmark suite, which consists of the following programs:

- sha: Compute the SHA-256 hash of a 484-character string.
- binom: Merge two 1000-element binomial queues and find the maximum element.
- color: Color a 156-node, 1168-edge graph using Kempe-Chaitin graph coloring.
- vs_easy, vs_hard: Decide validity of two separation logic entailments—one easy and one hard—using VeriStar [Stewart et al. 2012].

Figures 1 and 2 show the results for (partial) shrinking and uncurrying respectively, averaged across 10 runs. The comparison between our tool-derived partial shrinker and the hand-written shrinker is a bit apples-to-oranges: our partial shrinker doesn't perform dead function elimination and therefore does extra work on input terms with many dead function bodies. Nonetheless, our partial shrinker is at most 40% slower than the hand-written shrinker. Our generated uncurryer is about as fast as the hand-written one. As described in section 2.4, the uncurryer must make a recursive call on a term that is not a structural subterm of its input in order to perform all uncurrying in one pass; a nontrivial termination argument is needed to implement such a recursive function. We suspect that the differences in the way this argument is made are what cause the discrepencies in performance between the two implementations. We are not sure why the tool-derived uncurryer outperforms its hand-written counterpart on the vs_hard benchmark; we speculate that this could be due to the fact that the hand-written counterpart performs more allocation, and therefore incurs more calls to

[7]Since our generated inliner manages names in a fundamentally different way compared to the one currently in CertiCoq, it doesn't really have a hand-written counterpart; as such, it's not included in our measurements.

the garbage collector on this one test case.[8] While not definitive, these benchmarks suggest that tool-derived implementations are capable of being as efficient as hand-written counterparts.

6 DISCUSSION

So far we have described how our tool was used to implement the program transformations discussed in section 2. To demonstrate how it could be more broadly applicable, we briefly describe a few other program transformations on the λ_{cps} language that can be expressed in our framework.

6.1 Lambda Lifting

Lambda lifting is a transformation that reduces the overhead of function closures for functions with (at least some) known call-sites. If a function $f(x, y) = e$ has free variables a, b, then the lambda-lifted function $f'(x, y, a, b) = e$ can be used at applied occurrences of f, while escaping occurrences of f can use $f(x, y)$; the original definition's function body e can be replaced by $f'(x, y, a, b)$. This transformation can be expressed by the following top-down rule, which operates on lists of mutually recursive function definitions:

$$\frac{f', \vec{x}', \vec{y}' \text{ fresh} \qquad \vec{y} \subseteq \text{FV}(\vec{fd}_{\text{left}} +\!\!+ f(\vec{x}) = e :: \vec{fd}_{\text{right}})}{C[\textbf{let } \vec{fd}_{\text{left}} +\!\!+ (f(\vec{x}) = e) :: \vec{fd}_{\text{right}} \textbf{ in } e_1] \quad \longrightarrow \quad \begin{array}{l} C[\textbf{let } \vec{fd}_{\text{left}} \\ \quad +\!\!+ f'(\vec{x}' +\!\!+ \vec{y}') = \text{Rec } e[\vec{x}'/\vec{x}, \vec{y}'/\vec{y}] \\ \quad :: f(\vec{x}) = f'(\vec{x} +\!\!+ \vec{y}) \\ \quad :: \text{Rec } \vec{fd}_{\text{right}} \\ \quad \textbf{in } e_1] \end{array}}$$

The rule rewrites a function definition f in terms of a call to a lambda-lifted copy f'; f' takes additional parameters $\vec{y}'$ which close over some free variables $\vec{y}$ of the surrounding function bundle $\vec{fd}_1 +\!\!+ (f(\vec{x}) = e) :: \vec{fd}$, and its body e is accordingly rewritten in terms of the new variables. As with uncurrying, this transformation relies on future inlining passes to inline fully saturated calls to f, leaving behind calls to its lambda-lifted counterpart. Because lambda lifting is not always a beneficial transformation, a correct implementation need not close over all free variables, and is free to choose any subset $\vec{y}$ of the free variables of the mutually recursive block currently being traversed.

An efficient implementation requires:

- A state variable $\vec{fv} : \mathcal{P}(\text{Var})$ to hold the free variables of a mutually recursive block while traversing its function bodies, with invariant

$$\left\langle C \,\middle|\, \vec{fd}_{\text{right}} \right\rangle \sim \vec{fv} \iff \begin{array}{l} \exists\, D\, \vec{fd}_{\text{left}}\, e, \\ C[\vec{fd}_{\text{right}}] = D[\textbf{let } \vec{fd}_{\text{left}} +\!\!+ \vec{fd}_{\text{right}} \textbf{ in } e] \wedge \\ \vec{fv} = \text{FV}(\vec{fd}_{\text{left}} +\!\!+ \vec{fd}_{\text{right}}) \end{array}$$

- A delayed substitution just like the one described in section 2.2,
- A counter variable for fresh name generation, just like the one described in section 2.4.

Note this is not a linear-time algorithm—quadratic time is needed to compute free variable sets. However, our hand-written lambda lifter does this as well.

[8] To implement well-founded recursion, our tool invokes a fixed-point combinator that is difficult to work with manually, but carefully designed to extract to OCaml code with as little overhead as possible. Our manually-written uncurryer is written using the Equations library [Sozeau 2010], which makes it easy to write functions with custom termination metrics, but generates OCaml code that (for rather technical reasons) performs an extra allocation at each recursive call.

6.2 Dead Parameter Elimination

A parameter x of a (possibly recursive) function f is dead if it's only ever passed to dead parameters of f or other functions. Such parameters can be safely deleted. This can be accomplished by rewriting f as a wrapper around a copy f' that does not take any of its dead parameters as arguments and marking f for inlining. As a top-down rewrite rule:

$$\frac{\vec{y} \subseteq \vec{x} \qquad f' \text{ fresh} \qquad \vec{y} \text{ dead in } C[\mathbf{let}\ (f(\vec{x}) = e) :: \vec{fd}\ \mathbf{in}\ e]}{C[\mathbf{let}\ (f(\vec{x}) = e) :: \vec{fd}\ \mathbf{in}\ e] \quad \longrightarrow \quad C[\mathbf{let}\ (f'(\vec{x} \setminus \vec{y}) = e[f'/\vec{y}]) :: (f(\vec{x}) = f'(\vec{y})) :: \vec{fd}\ \mathbf{in}\ e]}$$

This rule rewrites f as a wrapper around f', and replaces all occurrences of dead variables $\vec{y}$ with f'. These bogus uses of f' will all be eliminated after all calls to wrapper functions have been inlined. (Our λ_{cps} language makes it inconvenient to, for example, supply a constant 0 in those positions, so we have to use *some* variable.)

To implement this rule efficiently, a counter variable is needed to generate fresh names, a delayed substitution is needed to perform the substitution $[f'/\vec{y}]$, and a state variable $\vec{d} : \mathcal{P}(\text{Var})$ is needed to record the names of all dead parameters. This state variable has invariant

$$\langle C \mid e \rangle \sim \vec{d} \iff \vec{d} \text{ dead in } C[e]$$

and can be initialized by a prepass that performs a standard dataflow analysis to determine which parameters can be safely eliminated.

6.3 Shrink Inlining

In addition to dead code elimination (§2.3), case folding (§2.1), and projection folding (§2.2), the full shrink reduction algorithm also inlines functions called only once. This so-called shrink inlining transformation can be specified by the following top-down rule:

$$\frac{|\mathbf{let}\ \vec{fd}_1 +\!\!+ (f(\vec{x}) = e) :: \vec{fd}_2\ \mathbf{in}\ D|_f = 0}{C[\mathbf{let}\ \vec{fd}_1 +\!\!+ (f(\vec{x}) = e) :: \vec{fd}_2\ \mathbf{in}\ D[f(\vec{y})]] \longrightarrow C[\mathbf{let}\ \vec{fd}_1 +\!\!+ \vec{fd}_2\ \mathbf{in}\ D[\text{Rec } e[\vec{y}/\vec{x}]]]}$$

where $|C|_f$ = the number of times f appears in context C in non-binding position

This difficulty with this rule is that it simultaneously inlines f and deletes its definition, and these two edits are made in very different places. Our theory doesn't permit rewrite rules which make edits in two places at once, so we must make do with specifying shrink inlining as a combination of two rewrite rules. These rules are (1) the dead function elimination rule described in section 2.3 and (2) the following top-down rewrite rule that inlines a function called once without deleting its definition:

$$\frac{|\mathbf{let}\ \vec{fd}_1 +\!\!+ (f(\vec{x}) = e) :: \vec{fd}_2\ \mathbf{in}\ D|_f = 0}{C[\mathbf{let}\ \vec{fd}_1 +\!\!+ (f(\vec{x}) = e) :: \vec{fd}_2\ \mathbf{in}\ D[f(\vec{y})]] \longrightarrow C[\mathbf{let}\ \vec{fd}_1 +\!\!+ (f(\vec{x}) = e) :: \vec{fd}_2\ \mathbf{in}\ D[\text{Rec } e[\vec{y}/\vec{x}]]]}$$

Appel and Jim [1997] use the delayed substitution σ described in section 2.2 to implement the substitution $[\vec{y}/\vec{x}]$, the occurrence count map δ described in section 2.3 to check if f is called exactly once, and an extra state variable $\theta : \mathcal{P}(\text{Var})$ to record which functions have been shrink-inlined and whose definitions should be deleted. The implementation is carefully written so that no functions mentioned in θ remain after shrink-reducing a block of function definitions.

Our tool does not provide such a strong guarantee: all it proves about derived implementations is that they refine a system of rewrite rules. It just so happens that, if holes are filled properly, our algorithm for deriving implementation from specification will produce an implementation that never forgets to delete shrink-inlined function definitions before exiting a block of function

definitions. However, our tool will not generate a proof of this fact. This complicates things: if we have no proof that an implementation deletes every shrink-inlined function definition before exiting a block of function definitions, then shrink inlined function bodies may be duplicated and one can no longer establish the unique binding property needed for delayed substitution to work properly.

Our solution is to split the variable θ into two halves $\theta_C, \theta_e : \mathcal{P}(\text{Var})$ with the invariant that using each half to drop the functions in the context and subterm in focus respectively yields a well-scoped term:

$$\langle C \mid e \rangle \sim (\theta_C, \theta_e) \iff \text{unique\_bindings}(P) \wedge \text{FV}(P) \cap \text{BV}(P) = \emptyset$$
$$\text{where } P = (\text{drop\_fns\_ctx } \theta_C\ C)[\text{drop\_fns } \theta_e\ e]$$

The implementation can then remove functions from θ_e as their definitions are deleted from the program. If θ_e is empty—which our tool-derived implementations ensure if all holes were filled in properly—then drop_fns θ_e $e = e$ and the term e satisfies the full unique binding property. Thus, even though we have no proof that our tool-derived implementation will delete every shrink-inlined function definition, we can verify this fact efficiently at runtime simply by checking that θ_e is the empty map.

This solution incurs some overhead in shuffling entries from θ_C to θ_e, deleting entries from θ_e after the corresponding function definitions have been deleted, and checking whether θ_e is empty before performing delayed substitution; however, these operations are all cheap to perform, and the resulting shrink reduction algorithm is about as fast as the one given by Appel and Jim.

6.4 Limitations

Some program transformations are not compatible with our framework. Notably, cross-language transformations (e.g. SSA conversion, code generation, and closure conversion) aren't easily expressible as rewrite systems. This is because a cross-language transformation must convert *every* source-language construct into appropriate target-language code. One could combine source and target together into a multi-language [Matthews and Findler 2007] and express the transformation as a set of rewrite rules on this multi-language, but our tool doesn't guarantee that the implementation it generates will eliminate all source-language constructs. Thus one would have to prove this fact by hand, reasoning about the very dependently typed tool-derived implementation.

7 RELATED WORK

Lacey and De Moor [2001] specify program transformations as rewrite rules of the form $\mathcal{I}_1 \rightsquigarrow \mathcal{I}_2$ if ϕ, where $\mathcal{I}_1$ and $\mathcal{I}_2$ are instructions in a control-flow graph and ϕ is a temporal logic formula representing some side conditions on the surrounding context. This approach has been further explored by Cobalt [Lerner et al. 2005a], its successor Rhodium [Lerner et al. 2005b], and PTRANS [Mansky 2014]. The main difference between this work and ours is in focus: all three of these systems present domain-specific languages for specifying program transformations based on temporal logic and develop algorithms that automatically prove such specifications sound; specifications are run using a rule execution engine that performs pattern matching on control-flow graphs. These systems either uses translation validation to ensure that rules are executed properly or require the engine to be a part of the trusted code base. We focus instead on efficiency and control: our theory allows implementors to specify precisely how they want dataflow facts computed, and we avoid the overhead of a rule execution engine and translation validation by generating plain recursive functions with a once-and-for-all refinement proof.

The Pilsner verified compiler [Neis et al. 2015] has its own framework for specifying program transformations; in this framework, a program transformation consists of (1) a pre-pass that

analyses the input term and marks selected subexpressions with to-be-performed rewrites, and (2) an implementation of each individual rewrite as a partial function. Only the individual rewrites need be proved sound; then, the two components are used to generate both an implementation that traverses the input term and applies individual rewrites at the locations indicated by the pre-pass and a proof that the implementation is sound. We develop this idea further by allowing fine-grained control over not only the locations where rewrites should be performed, but also the way in which dataflow facts should be computed and the delaying of intermediate computations that would otherwise require extra traversals.

In general, the use of a high-level language to specify program transformations is far from new: various domain-specific languages have been proposed over the years, including Gospel [Whitfield and Soffa 1997], Sharlit [Tjiang and Hennessy 1992], Scheme's macro-by-example [Kohlbecker and Wand 1987], the Nanopass framework [Sarkar et al. 2005], and Stratego [Visser 2004]. However, none of these systems permit control over the kinds of auxiliary data structures often needed for efficiency while retaining a formal guarantee of correctness with respect to a relational specification; moreover, most require learning a new specification language and system, while our rewriters can be specified as ordinary Coq inductive relations and implemented using ordinary Coq tactics.

Finally, an alternative approach to implementing program transformations is to learn them by example [Rolim et al. 2017] or from correctness proofs of more specific transformations [Tate et al. 2010], relieving implementors from having to think about their transformations in full generality. We believe that this approach could be used in conjunction with our own; so that a specification could be learned from more concrete examples via these methods and then implemented using our theory.

8 CONCLUSIONS AND FUTURE WORK

We have presented a theory that can specify a number of program transformations used by functional-language compilers, complete with auxiliary data structures needed for efficiency when implementating these transformations. We have described a procedure for deriving implementations from such specifications semi-automatically, along with a machine-checked proof of correctness. We implemented our procedure in Coq and used it to write inlining, uncurrying, and partial shrink-reduction passes. In some cases, our tool-derived implementations require significantly less manual proof effort to write and prove correct, and benchmarks suggest that they extract to efficient executable code.

In the future, we would like to add support for reasoning about the generated function in ways not related to the fact that it refines the transitive closure of a rewriting relation: for example, one might want to prove that an implementation reaches a fixed point in one pass, or that a state variable used to count the number of edits really does so. Currently, reasoning about the generated function in order to prove theorems like these is exceptionally challenging, as the function makes extensive use of dependent types and is cluttered by proof terms. One approach we are considering to mitigate this is to generate (and automatically prove) lemmas that characterize the behavior of the transformation on certain inputs, recovering the ability to use equational reasoning when proving properties about the function. This is the approach taken, for example, by the Equations library [Sozeau 2010].

Finally, we would like to implement and prove correct even more program transformations. Several other transformations are good candidates for implementation using our tool, including the ones discussed in section 6, eta expansion, hoisting, and loop preheaders [Appel 1994].

ACKNOWLEDGMENTS

This research was supported in part by National Science Foundation grants CCF-1521602 and CCF-2005545. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*.

Andrew W. Appel. 1994. Loop headers in λ-calculus or CPS. *Lisp and Symbolic Computation* 7, 4 (1994), 337–343.

Andrew W. Appel and Trevor Jim. 1997. Shrinking lambda expressions in linear time. *Journal of Functional Programming* 7, 5 (1997), 515–540.

Andrew W. Appel and David B. MacQueen. 1991. Standard ML of New Jersey. In *3rd International Symp. on Prog. Lang. Implementation and Logic Programming*, Martin Wirsing (Ed.). Springer-Verlag, New York, 1–13.

Nick Benton, Andrew Kennedy, Sam Lindley, and Claudio Russo. 2004. Shrinking reductions in SML. NET. In *Symposium on Implementation and Application of Functional Languages*. Springer, 142–159.

Pierre Chambart, Mark Shinwell, Leo White, and Damien Doligez. 2016. *Optimization with Flambda*. Chapter 21. https://caml.inria.fr/pub/docs/manual-ocaml/flambda.html

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (October 1991), 451–490.

Martin Erwig and Simon Peyton Jones. 2001. Pattern guards and transformational patterns. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 3.

Gérard Huet. 1997. Functional pearl. *J. functional programming* 7, 5 (1997), 549–554.

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

Eugene E Kohlbecker and Mitchell Wand. 1987. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 77–84.

D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. 1986. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices (Proc. Sigplan '86 Symp. on Compiler Construction)* 21, 7 (July 1986), 219–33.

Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices* 49, 1 (2014), 179–191.

David Lacey and Oege De Moor. 2001. Imperative program transformation by rewriting. In *International Conference on Compiler Construction*. Springer, 52–68.

Sorin Lerner, Todd Millstein, and Craig Chambers. 2005a. Cobalt: A language for writing provably-sound compiler optimizations. *Electronic Notes in Theoretical Computer Science* 132, 1 (2005), 5–17.

Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. 2005b. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. *SIGPLAN Not.* 40, 1 (Jan. 2005), 364–377. https://doi.org/10.1145/1047659.1040335

Xavier Leroy et al. 2012. The CompCert verified compiler.

William Mansky. 2014. *Specifying and verifying program transformations with PTRANS*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

Jacob Matthews and Robert Bruce Findler. 2007. Operational semantics for multi-language programs. *ACM SIGPLAN Notices* 42, 1 (2007), 3–10.

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: a compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 166–178.

Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.* 12, 5 (July 2002), 393âĂŞ434. https://doi.org/10.1017/S0956796802004331

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415.

Dipanwita Sarkar, Oscar Waddell, R Kent Dybvig, and EDUCATIONAL PEARL. 2005. A nanopass framework for compiler education. *Journal of Functional Programming* 15, 5 (2005), 653.

Olivier Savary Bélanger and Andrew W. Appel. 2017. Shrink fast correctly!. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*. 49–60.

Matthieu Sozeau. 2010. Equations: A dependent pattern-matching compiler. In *International Conference on Interactive Theorem Proving*. Springer, 419–434.

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* (2020), 1–53.

Guy L. Steele. 1978. *Rabbit: a compiler for Scheme.* Technical Report AI-TR-474. MIT, Cambridge, MA.

Gordon Stewart, Lennart Beringer, and Andrew W. Appel. 2012. Verified heap theorem prover by paramodulation. *ACM SIGPLAN Notices* 47, 9 (2012), 3–14.

Ross Tate, Michael Stepp, and Sorin Lerner. 2010. Generating Compiler Optimizations from Proofs. *SIGPLAN Not.* 45, 1 (Jan. 2010), 389–402. https://doi.org/10.1145/1707801.1706345

Steven WK Tjiang and John L Hennessy. 1992. Sharlit—a tool for building optimizers. *ACM SIGPLAN Notices* 27, 7 (1992), 82–93.

Eelco Visser. 2004. Program transformation with Stratego/XT. In *Domain-specific program generation.* Springer, 216–238.

Deborah L Whitfield and Mary Lou Soffa. 1997. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 6 (1997), 1053–1084.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. *ACM SIGPLAN Notices (PLDI'11)* 46, 6 (2011), 283–294.