Compositional Optimizations for CertiCoq

ZOE PARASKEVOPOULOU, Northeastern University, USA
JOHN M. LI, Princeton University, USA
ANDREW W. APPEL, Princeton University, USA

Compositional compiler verification is a difficult problem that focuses on separate compilation of program components with possibly different verified compilers. Logical relations are widely used in proving correctness of program transformations in higher-order languages; however, they do not scale to compositional verification of multi-pass compilers due to their lack of transitivity. The only known technique to apply to compositional verification of multi-pass compilers for higher-order languages is parametric inter-language simulations (PILS), which is however significantly more complicated than traditional proof techniques for compiler correctness. In this paper, we present a novel verification framework for *lightweight compositional compiler correctness*. We demonstrate that by imposing the additional restriction that program components are compiled by pipelines that go through *the same sequence of intermediate representations*, logical relation proofs can be transitively composed in order to derive an end-to-end compositional specification for multi-pass compiler pipelines. Unlike traditional logical-relation frameworks, our framework supports divergence preservation—even when transformations reduce the number of program steps. We achieve this by parameterizing our logical relations with a pair of *relational invariants*.

We apply this technique to verify a multi-pass, optimizing middle-end pipeline for CertiCoq, a compiler from Gallina (Coq's specification language) to C. The pipeline optimizes and closure-converts an untyped functional intermediate language (ANF or CPS) to a subset of that language without nested functions, which can be easily code-generated to low-level languages. Notably, our pipeline performs more complex closure-allocation optimizations than the state of the art in verified compilation. Using our novel verification framework, we prove an end-to-end theorem for our pipeline that covers both termination and divergence and applies to whole-program and separate compilation, even when different modules are compiled with different optimizations. Our results are mechanized in the Coq proof assistant.

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

Additional Key Words and Phrases: compositional compiler correctness, logical relations, separate compilation, compilation by transformation, A-normal form, closure conversion, lambda lifting

ACM Reference Format:
Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. 2021. Compositional Optimizations for CertiCoq. *Proc. ACM Program. Lang.* 5, ICFP, Article 86 (August 2021), 30 pages. https://doi.org/10.1145/3473591

1 INTRODUCTION

The state of the art in compiler verification is devising proof techniques that support not only whole-program compilation but also separate compilation of program components with the same or different compilers. The specification of a compiler is typically expressed as a relation between a source and a target program, $R\ (e_{\text{src}}, e_{\text{trg}})$. For R to be a meaningful compiler specification, it must be an *adequate* relation—it must imply that the target program refines the behavior of the source

Authors' addresses: Zoe Paraskevopoulou, Northeastern University, USA, z.paraskevopoulou@northeastern.edu; John M. Li, Princeton University, USA, johnli@princeton.edu; Andrew W. Appel, Princeton University, USA, appel@princeton.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/8-ART86
https://doi.org/10.1145/3473591

program. A compiler specification is compositional when it is *compatible with linking*. That is, if $R\ (e_{\text{src}}, e_{\text{trg}})$ and $R\ (e'_{\text{src}}, e'_{\text{trg}})$ then $R\ ([e'_{\text{src}}]e_{\text{src}}, [e'_{\text{trg}}]e_{\text{trg}})$, where $[e']e$ denotes linking programs e and e'. This property is also called *horizontal compositionality*. Relations that are both adequate and compatible with linking allow us to prove that the whole program obtained by linking two separately compiled programs in the target language refines the behavior of the whole program obtained by linking the programs in the source language.

Developing relations that are both adequate and compatible with linking is a hard problem and the subject of growing research [Hur et al. 2012; Neis et al. 2015; Ramananandro et al. 2015; Stewart et al. 2015; Patterson and Ahmed 2019; Song et al. 2019]. Logical relations are widely-used as a technique for showing program equivalence and refinement, and they have long been used in compiler correctness [Benton and Hur 2009; Hur and Dreyer 2011; Hur et al. 2012; Perconti and Ahmed 2014; RodrÃŋguez et al. 2016; Owens et al. 2017]. Logical relations are preserved under linking and therefore they have *horizontal* compositionality. But logical relations lack *vertical* compositionality: the correctness theorems of adjacent transformations that are independently proved correct cannot be composed to derive a top-level theorem for the whole pipeline. Therefore, they do not scale to compositional correctness for multi-pass compilers. To address this limitation, Neis et al. [2015] devise parametric inter-language simulations (PILS), a novel relation for compositional compiler correctness that combines logical relations and bisimulations and is both adequate and compatible with linking. However, PILS—the only relations known to apply to compositional compiler verification for higher-order, functional languages—are much more complicated than traditional logical relations and involve quite complex metatheory.

In this paper, we propose a new *lightweight* technique for compositional compiler correctness for higher-order functional languages. We demonstrate that if we restrict ourselves to compilers that go through *the same intermediate representations*, we can use step-indexed logical relations to build a top-level compiler relation that has both vertical and horizontal compositionality. Our key insight is that adequacy and compatibility with linking are properties closed under relation composition. Therefore, we define our top-level relation to be the composition of the logical relations that are used to verify adjacent compiler passes. This top-level relation is adequate, compatible with linking, and independent of the transformations that are used—it only depends on the changes of representation through compilation. Pipelines that go through the same changes of representation can be given the same specification regardless of the particular transformations that are used. As a corollary of this top-level compositional theorem, we can derive that compiling modules with different verified pipelines and linking them in the target produces the same behavior as linking the modules in the source language and compiling them as a whole. Our compositional verification approach can be applied to standard (and relatively simple), untyped, step-indexed logical relations. Such relations have been used in whole-program compiler verification before [Owens et al. 2017].

The top-level refinement relation that we establish for the compiler and each individual transformation is in the form of a *forward simulation*: we show that whenever the source exhibits a behavior—which can be either termination or divergence—then the target exhibits the same behavior. This is an appropriate compiler specification since we are in a deterministic language setting.[1] But logical relations only specify what happens when the source program terminates, and they do not capture diverging source program executions. Building on a previous technique for reasoning about resource bounds [Paraskevopoulou and Appel 2019], we show how to apply logical relations to show that divergence is preserved. We achieve that by using a big-step, fuel-based

[1]Compilers are typically specified with a backwards simulation: every behavior that is exhibited by target program should be a valid behavior of the source program. In deterministic languages, forward simulations are often preferred as they are simpler to prove and they imply backwards simulations. For a detailed discussion see Leroy [2009b].

semantics and parameterizing our logical relation with a *relational invariant*, a binary relation that relates the fuel consumption of the source and target programs. We then show that for an appropriate class of relational invariants the logical relation implies that divergence is preserved.

A similar approach is used in the verified CakeML compiler. CakeML's forward simulations establish that the fuel consumption of the source never exceeds that of the target; this implies divergence preservation. But this condition is not satisfied by transformations that reduce the number of program steps. Therefore, CakeML introduces (in each intermediate language) a `Tick` instruction that has no computational meaning other than consuming fuel. Transformations will insert `Tick` instructions as needed to keep the fuel consumption of the target equal that of the source program. This approach requires a `Tick`-removal pass that must be proved correct with a backwards simulation. In our framework, we show that a more general relation between fuel consumption suffices to show divergence preservation and hence, we do not need to add a `Tick` instruction to our language.

We apply this framework to verify the multi-pass optimizing middle-end pipeline of CertiCoq [Anand et al. 2017]. CertiCoq is a compiler from Gallina (Coq's specification language) to machine language. Its front end uses MetaCoq to reify Coq functions and erase types with a proved-sound transformation [Sozeau et al. 2019]. Constructor applications are then converted to η-long form. The program is CPS-converted [Paraskevopoulou and Grover 2021] or ANF-converted into the intermediate language λ_{ANF}, a higher-order, functional intermediate representation in an A-normal form [Flanagan et al. 1993]. CertiCoq's optimizer and closure-converter translates to $\lambda^{\mathsf{C}}_{\mathsf{ANF}}$, a closureless, flat-scope subset of λ_{ANF}. Lastly, there is a proved-correct translation [Savary Bélanger et al. 2019] to CompCert Clight [Leroy 2009a]. Any C compiler can be used as the back end, but only CompCert can yield an end-to-end correctness guarantee.

Our λ_{ANF} pipeline follows a *compilation by transformation* approach [Kelsey and Hudak 1989; Fradet and Le Métayer 1991; Jones and Santos 1995]: through the interaction of small and modular transformations it optimizes the code and gradually compiles away language features that cannot be mapped directly to the target language. The pipeline comprises 7 distinct transformations: shrink reduction [Appel and Jim 1997a; Benton et al. 2005], inlining, uncurrying, lambda lifting, hoisting, and dead-parameter elimination. Similar transformations have been implemented and described in other compilers [Adams et al. 1986; Steele 1978; Appel 1992; Jones 1996; Kennedy 2007; Leroy et al. 2020], where it is observed that simplifications performed by one phase can "cascade", enabling more simplifications by the same or other phases. Through the interaction of these passes our pipeline achieves efficient *closure allocation strategies*, avoiding useless heap allocation of function closures. To our knowledge, CertiCoq's closure allocation strategies are the most advanced among formally verified compilers. Our preliminary experimental results show that CertiCoq with our λ_{ANF} pipeline runs at native-code speed and that the performance of the code is improved by our closure allocation optimizations. Our verification framework is particularly suitable for the design of the pipeline. New transformations can be easily added (and proved correct with the same machinery) and old transformations can be reordered without any effect on the compositional top-level theorem. Our separate compilation theorem applies to programs compiled with different versions of the pipeline.

Contributions. In summary, we present:

- A novel lightweight technique for compositional compiler correctness. The technique is compatible with standard logical relations and supports compositional verification of different pipelines that go through the same representation changes.
- A novel method for proving divergence preservation using logical relations, compatible with optimizations that reduce the number of program steps and compositional compiler correctness.

- λ_{ANF}, an optimizing, modular, general-purpose, extensible, compilation pipeline for pure, higher-order functional languages, verified with the above techniques. Our pipeline implements more sophisticated closure strategies than previous verified compilers.

Our results are fully mechanized in the Coq proof assistant. We begin with a language-generic overview of our novel compositional verification approach and our technique for showing divergence preservation.

2 OVERVIEW OF THE VERIFICATION APPROACH

2.1 Compositional Reasoning

Step-indexed logical relations [Appel and McAllester 2001; Ahmed 2006] are a useful tool for proving a variety of properties about lambda calculus and related languages. Binary logical relations are commonly used to show program equivalence and refinement [Benton and Hur 2009; Perconti and Ahmed 2014; Owens et al. 2016, 2017; Timany et al. 2017], making them a useful tool for compiler verification. A particularly desirable property of binary logical relations in compiler verification [Benton and Hur 2009; Hur and Dreyer 2011] is horizontal compositionality: two pairs of language expressions that are *independently* shown to inhabit the logical relation will still inhabit the logical relation when they are pairwise composed with the same language construct. For example, if for a logical relation R and for two pairs of expressions we have that $R(e_1, e_2)$ and $R(e_1', e_2')$, then we can show that applying e_1 to e_1' and e_2 to e_2' yields related expressions, *i.e.*, $R(e_1\ e_1', e_2\ e_2')$. This is established by the so-called compatibility lemmas of the logical relation.

Since linking is typically defined using language constructs, logical relations are compatible with linking. This is a crucial property for compositional compiler correctness: two pairs of logically related programs of arbitrary provenance can be linked together to obtain two logically related programs. For example, assume that compilers comp_1 and comp_2 are independently verified to satisfy a logical relation R: $\forall\ e,\ R(e, \mathsf{comp}_i(e))$, with $i \in \{1, 2\}$. Then, for any programs e_{lib} and e_{client}, we can derive that $R\ ([e_{\mathrm{lib}}]e_{\mathrm{client}}, [\mathsf{comp}_1(e_{\mathrm{lib}})]\mathsf{comp}_2(e_{\mathrm{client}}))$, where $[e_{\mathrm{lib}}]e_{\mathrm{client}}$ denotes linking the library program e_{lib} with the client program e_{client}.

Unfortunately, cross-language logical relations[2] are not transitively (or vertically) composable. That means, that if R_{1-2}, R_{2-3}, and R_{1-3} are logical relations between languages L_1 and L_2, L_2 and L_3, and L_1 and L_3, one *cannot* derive that $\forall\ e_1\ e_2\ e_3,\ R_{1-2}\ (e_1, e_2) \Rightarrow\ R_{2-3}\ (e_2, e_3) \Rightarrow\ R_{1-3}\ (e_1, e_3)$ Therefore, logical relations cannot be used to prove compositional specifications for multi-pass compilers. To understand this, consider the definition of logical relations for function values: two functions are logically related if they map logically related values to logically related results. To show that $R_{1-3}\ (f_1, f_3)$ given $R_{1-2}\ (f_1, f_2)$ and $R_{1-3}\ (f_2, f_3)$ for functions f_1, f_2, and f_3, one would have to show that given arguments v_1 and v_3 such that $R_{1-3}\ (v_1, v_3)$, R_{1-3} relates the expressions $f_1\ v_1$ and $f_3\ v_3$. But the proof is stuck: there is no way to use the hypotheses since there is no value v_2 such that $R_{1-2}\ (v_1, v_2)$ and $R_{2-3}\ (v_2, v_3)$.

To address this limitation, Hur et al. [2012] develop parametric bisimulations (PBs), a new relation that combines ideas from Kripke logical relations and bisimulations to derive a relation that is both horizontally and vertically composable. Later, Neis et al. [2015] develop parametric inter-language simulations (PILS) that generalize PBs to a cross-language setting. Roughly, PBs and PILs work by parameterizing the relation with an unknown, arbitrary relation, the "assumed equivalence" and drawing function arguments from this relation. But applying related functions to "assumed equivalent" arguments is not guaranteed to produce the same observable behavior since nothing is

[2]Same-language logical relations may or may not be transitive. Typed same-language logical relations can be made transitive by imposing additional requirements about typing [Ahmed 2006]. Some untyped logical relations are transitive [Owens et al. 2017], others, like the ones we will use in these paper, are not.

known about assumed equivalence. The solution is to define a local term equivalence relation that asserts that two terms have the same observable behavior, excluding what happens in calls related by assumed equivalence. These definitions are more convoluted than standard logical relations used in compiler correctness (*e.g.*, the one in [Owens et al. 2017]). Even though they avoid the usual step-indexing that is needed to make logical relations well-founded, they need to be coinductively defined and hence, proofs require coinductive reasoning. As those authors say in their papers, the metatheory of PBs and PILS (including the proof of transitivity) are significantly complicated. For example, a very subtle point in the transitivity proof is decomposing the assumed equivalence between e_1 and e_3 to derive assumed equivalences between e_1 and e_2 and e_2 and e_3. Furthermore, in their original form, PBs and PILS do not admit $\eta-$conversion (which many transformations perform) and further adjustments are needed to make the rule admissible [Hur et al. 2014].

In this work, we show that if we are willing to restrict our compositional specification to only apply to compilers that encompass the same sequence of intermediate representations, standard logical relations can be used to derive a compositional specification for multi-pass compilers. We observe that adequacy and horizontal compositionality are closed under relation composition, under the assumption that program refinement is transitively composable (which is true in our setting and in many other settings as well [Leroy 2009b; Chlipala 2010; Tan et al. 2016]). This follows immediately by the definition of horizontal compositionality, adequacy, and relation composition.

Therefore, we can obtain a top-level compositional relation by composing the logical relations that are used to verify each pass. For example, we might have compilers that go through languages L_1, L_2 and L_3 with two passes each that are verified with logical relations R_{1-2} and R_{2-3}. We define a top-level relation $\overline{R} \overset{\text{def}}{=} R_{1-2} \circ R_{2-3}$. Then, it is easy to show that $\overline{R}$ is adequate and compatible with linking.

The above top-level relation will work for any compilers that have two passes, one between L_1 and L_2, and one between L_2 and L_3, that satisfy R_{1-2} and R_{2-3} respectively. However, compilers often perform one or more same-language transformations on the same intermediate representation. We may want to verify compilers that include an arbitrary number of same language transformations on language L_2 that satisfy a same-language logical relation R_{2-2}. To that end, we define the following relation:

$$R^{+}_{2-2} \overset{\text{def}}{=} \{\ (e_1, e_2) \mid \exists\ n \geq 1,\ (e_1, e_2) \in \underbrace{R_{2-2} \circ \ldots \circ R_{2-2}}_{n\text{-times}}\ \}$$

Because untyped, same-language logical relations, like R_{2-2}, are reflexive,[3] R^{+}_{2-2} is satisfied by zero or more L_2 transformations. Therefore, the top-level relation

$$\overline{R} \overset{\text{def}}{=} R_{1-2} \circ R^{+}_{2-2} \circ R_{2-3}$$

is satisfied by compilers that encompass a pass from L_1 to L_2 that satisfies R_{1-2}, (possibly) an arbitrary number of L_2 passes that satisfy R_{2-2}, and a pass from L_2 to L_3 that satisfies R_{2-3}. As we will explain, the proof of horizontal compositionality for R^{+}_{2-2} makes use of reflexivity.

We apply this approach to prove correct our multipass λ_{ANF} pipeline. In our pipeline, only one representation change happens (by our closure conversion transformation) and the rest of the passes are same-language transformations before and after closure conversion. In our setting, we only need two logical relations: a cross-representation logical relation (for closure conversion that changes the representation of function values), and a same-language, reflexive logical relation (for all other same-language transformations). The benefit of this approach is that each pass can be

[3]Typed same-language logical relations are also reflexive but for well-typed programs; this property is usually called the fundamental property of the logical relation [Ahmed 2006]. In principle, we could apply our technique to typed logical relations but that would require proving that transformations preserve typing.

proved correct with a standard logical relation that has well-understood metatheory. Then the compositionality framework can be built on top of the logical relations without any changes to the logical-relations machinery. We found that defining the top-level compositional compiler relations and proving adequacy and compatibility with linking was mostly straightforward. The definitions of the top-level relations and the proofs of adequacy and compatibility (including the top-level behavioral refinement) comprise 319 lines of code and 490 lines of proof.

2.2 Divergence Preservation

Correct compilers are expected to preserve divergent behaviors: if the source program infinite-loops, then the target program should not return a result. Another contribution of our verification framework is a general technique to show that divergence is preserved using a forward simulation between big-step, fuel-based semantics. This is a delicate verification problem, especially when transformations reduce the number of program steps. Existing techniques apply to simulations that use small-step or big-step semantics.

Showing divergence preservation with a forward simulation between small-step semantics requires reasoning about stuttering steps [Leroy 2009b; Barthe et al. 2019]. The problem is that a small-step simulation for a step-reducing transformation states that whenever the source takes a step ($e_{\mathrm{src}} \rightarrow e'_{\mathrm{src}}$), then the target may take *zero* or more steps ($e_{\mathrm{trg}} \rightarrow^{*} e'_{\mathrm{trg}}$). This, however, does not exclude the possibility of the source program taking an infinite reduction sequence while the target program terminates to a value. This is called the stuttering problem. The standard solution is to define a well-founded measure on program states and show that it is strictly decreasing when a source step does not correspond to a target step. However, small-step semantics complicate semantic preservation proofs for transformations that introduce administrative redexes [Plotkin 1975; Dargaye and Leroy 2007] (our closure conversion, uncurrying, and lambda lifting all introduce administrative redexes). Therefore, big-step semantics are often favored in semantics preservation proofs. But big-step semantics do not usually capture terminating behaviors. One solution is to do a separate proof using a big-step coinductive judgment that captures divergent behaviors [Leroy and Grall 2009]. But this tends to be rather complicated and as requires formalizing two judgments and carrying out a separate semantics preservation proof for each of them. To avoid the complications of both small-step semantics and coinductive big-step semantics, the CakeML compiler uses a fuel-based big-step semantics [Owens et al. 2016, 2017] that, just like ours, can capture divergent behaviors. But to show divergence preservation, CakeML proofs require that the compiled program consumes at least as much fuel as the source program. Since this is not true for all transformations, CakeML uses a special instruction, `Tick`, in all of its intermediate representations that decreases the amount of available fuel. Transformations introduce a `Tick` instruction whenever steps are compiled away to keep the fuel consumption of the source upper bounded by the fuel consumption of the target program. The instruction is propagated through all intermediate languages and it is erased before code generation. It complicates correctness proofs by requiring an extra `Tick`-removal pass that has to be proved correct with a backwards simulation [Owens et al. 2017, Section 7].

In our work, we take a more general approach: we weaken the $c_{\mathrm{src}} \leq c_{\mathrm{trg}}$ relation that is enforced by CakeML to the fuel consumption of the source and target programs to a relation $c_{\mathrm{src}} \leq f(c_{\mathrm{trg}})$, where f can be any strictly monotonic function. Therefore, we allow the target to consume less fuel than the source program and we do not need to add `Tick` instructions to our intermediate representations.

First, we define a fuel-based big-step evaluation judgment, written $e \;{}^{c}\!\Downarrow r$, that asserts that with some fuel value c, the evaluation of an expression e yields a result r. Each evaluation rule consumes a unit of fuel. The result can be either a value or an out-of-time exception (OOT) if there is not enough fuel to evaluate the expression. We say that a program diverges if for all fuel values the evaluation of a program times out: $e \Uparrow \overset{\text{def}}{=} \forall\, c, e\ {}^{c}\!\Downarrow \mathsf{OOT}$.

We formally prove (for our λ_{ANF} semantics in section 4) that in order to show preservation of divergence, it is enough to show that when the source program times out with some fuel c_{src}, then so does the target program with some fuel c_{trg}, such that $c_{\text{src}} \le f(c_{\text{trg}})$ for some strictly monotonic function f. In order to prove this more general bound, which is typically different for each transformation, we use a logical relation that is parameterized by relational invariants on the fuel consumption of the two programs. The compatibility lemmas of the logical relation are adjusted so that we can compositionally prove that the invariants hold.

More precisely, we define a step-indexed logical relation is parameterized by two relational invariants: Q_L the *local* invariant, and Q_G the *global* invariant:

$$\begin{aligned}\mathcal{E}^k(e_1, e_2)\ \{Q_L; Q_G\} \ \overset{\text{def}}{=}\ & \forall\, c_1\, r_1,\ e_1\ {}^{c_1}\!\Downarrow r_2 \Rightarrow \\ & \exists\, c_2\, r_2,\ e_2\ {}^{c_2}\!\Downarrow v_2\ \wedge\ Q_L\ (c_1, c_2)\ \wedge\ \mathcal{R}^{k-c_1}(r_1, r_2)\ \{Q_G\}\end{aligned}$$

The logical relation asserts, as usual, that if expression e_1 evaluates to a result r_1 for any fuel value c_1 that is less than the step-index, then the target expression e_2 evaluates with some fuel value c_2, to some result r_2 such that the two results are related with a result relation. Crucially, it enforces that the fuel values of the two executions are related with the local invariant Q_L. The resulting relation relates results that are either both out-of-time exceptions or both related values. It is parameterized by the global invariant Q_G. This is more subtle; we defer its explanation to section 5 where we give a concrete definition of the result relation.

By showing that a transformation, trans, inhabits the relation for an appropriate choice of local invariant we show that divergence is preserved. That is, we must prove $\forall\, e,\ \mathcal{E}^k(e, \mathsf{trans}(e))\ \{Q_L; Q_G\}$ for some Q_L that implies that when $Q_L\ (c_1, c_2)$, then $c_1 \le f(c_2)$ for some strictly monotonic f. We found that such an upper bound was quite hard to find for the inlining transformation (but quite easy to establish with our logical relation once found). It required us to tweak our semantics to keep track of the total number of function calls that are executed by the program in addition to the fuel consumption.

This approach is inspired by that of Paraskevopoulou and Appel [2019] who show (among other things) that a closure conversion transformation preserves diverging behaviors. In particular, they use a logical relation parameterized by pre- and postconditions—which are analogous to our relational invariants—to impose a lower bound on the steps of the closure-converted program. However, they do not show how this technique can be applied to transformations that reduce the number of program steps.

In the next section, we describe our the intermediate representation and the multi-pass pipeline on which we apply these verification technique. In section 5, we set up our verification framework that incorporates these ideas.

3 THE λ_{ANF} PIPELINE

Our λ_{ANF} pipeline efficiently compiles away first-class functions and transforms the code to a representation that can be readily compiled to a first-order, low-level language. The pipeline will create a *known* and an *escaping* instance for each function [Appel 1992]. Known instances are used only in statically known function calls, whereas escaping instances are used when a function escapes to a different scope either through function return or parameter passing. Known instances can be compiled more efficiently using specialized calling conventions. Known curried functions can be uncurried by compiling them to functions that receive all of their arguments simultaneously. In addition, closure records of known functions can be eliminated and their closure environments can be allocated either in a heap-allocated environment or in registers. The latter is achieved with a lambda-lifting transformation that transforms functions to receive their free variables as parameters. In this section we formally define the λ_{ANF} intermediate representation and we give an overview of the λ_{ANF} pipeline.

3.1 The λ_{ANF} Intermediate Language

The λ_{ANF} language (fig. 1) is a lambda-calculus in A-normal form [Flanagan et al. 1993], extended with constructors, projections, case analysis and (mutually) recursive function definitions. There are two kinds of function calls: let-bound function application, where the control returns to the caller, and tail call. The patterns in case-analysis statements discriminate only the constructor and do not bind any arguments; arguments of constructors must be projected explicitly in the body of each case-clause. We use $\overline{\text{overbar}}$ to indicate repetition, so $\mathtt{fun}\ \overline{f\ \overline{x} = e_1}\ \mathtt{in}\ e_2$ defines a list of zero or more mutually recursive functions, each of which may have zero or more parameters. We use metanotation for operations on such lists: [] for empty list, $x :: \overline{x}$ for consing an element at the beginning of a list, $\overline{x} +\!\!+ \overline{y}$ for list concatenation, and $|\overline{x}|$ for list length.

$$
\begin{array}{llll}
x, y, f \in \mathtt{Var} & & & \text{Variables} \\
\mathtt{C} \in \mathtt{Constr} & & & \text{Constructors} \\
\\
e \in \mathtt{Exp} & ::= & \mathtt{let}\ x = \mathtt{C}(\overline{y})\ \mathtt{in}\ e & \text{Constructor} \\
 & | & \mathtt{let}\ x = y.i\ \mathtt{in}\ e & \text{Projection} \\
 & | & \mathtt{case}\ y\ \mathtt{of}\ [\mathtt{C}_i\ \to\ e_i]_{i \in I} & \text{Case} \\
 & | & \mathtt{fun}\ \overline{f\ \overline{x} = e_1}\ \mathtt{in}\ e_2 & \text{Function def.} \\
 & | & \mathtt{let}\ x = f\ \overline{y}\ \mathtt{in}\ e & \text{Function call} \\
 & | & f\ \overline{x} & \text{Tail call} \\
 & | & \mathtt{ret}(x) & \text{Return}
\end{array}
$$

Fig. 1. The λ_{ANF} language.

The input λ_{ANF} program may have nested functions with free variables. The output program is in a closure-converted subset $\lambda^{\mathsf{C}}_{\mathsf{ANF}}$. In this restricted subset, programs have the form $\mathtt{fun}\ \overline{f\ \overline{x} = e_f}\ \mathtt{in}\ e$, where all the functions are *closed*, and neither e nor the function bodies e_f contain function definitions. There are only two levels of scope in the program: the scope that contains the global definitions of the mutually defined functions and the local scope of each function body. In $\lambda^{\mathsf{C}}_{\mathsf{ANF}}$, function values can be represented as bare function pointers and do not require closures. In addition, all intermediate results are explicitly evaluated and stored in variables. Therefore, $\lambda^{\mathsf{C}}_{\mathsf{ANF}}$ is straightforward to code-generate to low-level languages such as C or assembly.

3.2 λ_{ANF} Transformations

The λ_{ANF} pipeline consists of 13 passes: Shrink-Uncurry-Inline-Shrink-λLift-Inline-Shrink-ClosureConvert-Hoist-Inline-Shrink-DPE-Shrink, built from the following 7 transformations.

Shrinking is a transformation that performs shrink reductions [Appel and Jim 1997b]: case and projection folding (*i.e.*, static evaluation of case statements and projections of values that are statically known), dead code elimination, and inlining of functions that are called exactly once. We use the proved-correct shrink-reducer of Savary Bélanger and Appel [2017] which operates on the CPS subset of our λ_{ANF}. We extended the implementation and the proof to apply to the full λ_{ANF} language and we adapted the proof to our verification framework. The previous proof of the shrink reducer showed semantic preservation only for terminating programs.

Inlining inlines nonrecursive function calls picked based on a heuristic or marked for inlining by a different pass (uncurrying and lambda lifting). Our current heuristic is to inline small-bodied functions. Inlining let-bound function calls suffers from the usual problems of inlining in A-normal form languages [Kennedy 2007]. When inlining a let-bound function call, our inliner will perform a renormalization step so that the program is in A-normal form. Inlining let-bound function calls whose body is a case expression requires introducing a join point [Maurer et al. 2017]. Our current inliner will not inline such functions, but this is not a fundamental limitation of our approach.

Uncurrying. After ANF conversion, all functions are unary (or 2-ary if CPS conversion is used, which adds an additional continuation argument). Multi-argument functions in Coq are curried: they receive an argument and return a new function until all of the arguments are consumed, causing a closure allocation for each intermediate function return. Our uncurrying transformation converts calls to statically known curried functions into efficient multi-argument calls. It does so by detecting curried functions based on their syntactic structure and introducing a new multi-argument function. The approach is described by Appel [1992].

Closure conversion and hoisting convert the input program to a program with flat function structure. Closure conversion eliminates free variables in functions by introducing explicit closures. Then a separate pass hoists all functions into a top-level block of mutually recursive closed function definitions. After this pass, all variable references are either references to function parameters and local variables or references to function names defined in the top-level function block.

Our closure-conversion transformation treats all functions uniformly. It closure-converts *every* function by installing a new environment parameter and creating a heap-allocated closure. The rest of the λ_{ANF} transformations eliminate redundant closures: lambda lifting creates known and escaping instances and (selectively) converts known functions to receive their free variables as parameters, shrink-reduction statically evaluates projections from statically known closure pairs, dead parameter elimination removes useless environment parameters.

Lambda Lifting [Hughes 1982; Johnsson 1985] runs before closure conversion and is key to the implementation of our efficient closure allocation strategies. This transformation creates closed functions by passing free variables as extra arguments so that they do not have to be passed through a closure environment. Effectively, this stores known-function closure environments in registers (instead of allocating them in the heap). Lambda lifting typically lifts closed function definitions to the top-level (hence its name); but we defer that step to our hoisting transformation. Our lambda lifting can be expressed as the congruent, transitive closure of the following local rewrite step.

$$\mathtt{fun}\ f\ \overline{x} = e_1\ \mathtt{in}\ e_2 \quad \rightsquigarrow \quad \begin{array}{l} \mathtt{fun}\ f'\ (\overline{fv} + \overline{x}) = \\ \quad \mathtt{fun}\ f\ \overline{x} = f'\ (\overline{fv} + \overline{x})\ \mathtt{in}\ e_1 \\ \mathtt{in} \\ \mathtt{fun}\ f\ \overline{x} = f'\ (\overline{fv} + \overline{x})\ \mathtt{in}\ e_2 \qquad \text{where } \overline{fv} \subseteq \mathsf{fv}(\mathtt{fun}\ f\ \overline{x} = e_1) \end{array}$$

It defines a known copy f' of the function f that receives its free variables as parameters. The function f is then redefined in terms of f' in both the local scope of the function and after the function definition. Known calls to f inside e_1 or e_2, will be inlined to call f'. When the function escapes, f' is called through the escaping wrapper f. Recursive calls will always call the known function instance f'. The known and the unknown instances are not mutually defined, because mutually defined functions share their closure environment. Therefore, the closure environment of the known instance would not become eliminated.

Our lambda lifting transformation is *selective*: not every known function instance will receive its free variables as parameters. As observed in other compilers [Santos 1995; Jones 1996] [Leroy et al. 2020, Chapter 21], lambda lifting every function indiscriminately may result in worse performance. We devise a new set of criteria for lambda lifting that, according to our preliminary experimental results, avoid worsening performance in the set of benchmarks we consider. Furthermore, selective lambda lifting achieves considerable performance improvement (6%-7%) in some cases. The criteria are as follows:

(1) The total number of a function's arguments after lambda lifting should not exceed the available registers in the machine in order to avoid register spilling.
(2) When a nontail call is performed, parameters and locals in (caller-save) registers must be pushed into the stack before the call and popped after it. If a free variable is used only late in a function-body, *after* such calls, it would be fetched late from the closure environment and no such pushing and popping would have occurred. But when such a free variable becomes a parameter (live across several calls), this generates more memory traffic than fetching its value from a heap-allocated closure environment. So we lambda-lift free variables only if they are live across at most one function call. To our knowledge, we are the first to make this observation about the performance of lambda lifting. As we demonstrate in our evaluation section, lambda lifting without taking this criterion into account does not give as good performance improvement.
(3) When a lambda-lifted function is called, the extra arguments can bloat the size of the calling function's closure if these variables are not part of the caller's local variables or closure environment. Our lambda lifting will (optionally) not inline calls to escaping function wrappers if that would cause an increase the size of the caller's closure environment. Note that this heuristic is off by default. Often there is a cascading effect and the closure of the caller can be avoided too, so this strategy can miss opportunities for optimization.

We believe these are the most refined lambda lifting strategies implemented by any compiler. Flambda's lambda lifter is not selective, and GHC's Core-to-Core (selective) lambda lifter [Santos 1995; Jones 1996; Graf and Jones 2019] only considers strategy (3). We also considered inlining the known function call inside the escaping wrapper (as OCaml's Flambda [Leroy et al. 2020, Chapter 21] optimizing pipeline does) but we did not observe performance improvement—probably because the back end implements tail calls to known functions as efficient jumps with arguments, or even by fall-through.

Dead parameter elimination removes unused arguments from functions. It performs liveness analysis around (mutually) recursive functions to find which arguments are truly needed by the computation, and it marks other arguments as unused. Some of the unused arguments will be useless closure environments installed by closure conversion: if (before closure conversion) a function is closed and has only known calls, then no closure environment is needed. Rather than adding this special case to our closure-conversion algorithm, we just build the closure anyway, and then clean it up with DPE and shrink-reduction.

3.3 Compilation by Example

We illustrate the compilation of a program through the λ_{ANF} pipeline with an example. We deviate slightly from the syntax of λ_{ANF} in order to keep the code shorter and more readable. In the following program, the function `interleave` traverses a list interleaving the free variables `x` and `y` among its elements. Then the function is applied to list `l1` and it is also applied to each element of the list (of lists) `l` using the `map` function.

```
fun interleave l =
  case l of
  | [] => []
  | z :: l => x::y::z::(interleave l)
  end in
```

```
let l1 = [1, 2 , 3] in let l = [l1, l1, l1] in
let l1' = interleave l1 in
let l' = map interleave l in ...
```

In the above program, uncurrying has already been done, and the known function `map` (assuming that it is defined earlier in the program) is applied to both of its arguments simultaneously. Next, lambda lifting is applied. It creates two copies of `interleave`. The known copy, `interleave_known`, receives its free variables as arguments and is a closed function. The escaping copy, `interleave`, is defined both inside the function body and after the function definition and will be used at escaping positions. Our inliner will inline statically known `interleave` calls. The shrinker will eliminate the dead code (*e.g.*, the definition of `interleave` inside `interleave_known`). After these transformations, we have:

```
fun interleave_known l x y =
    case l of
    | [] => []
    | z :: l => x::y::z::(interleave_known l x y)
    end in
fun interleave l = interleave_known l x y in
let l1 = [1, 2 , 3] in let l = [l1, l1, l1] in
let l1' = interleave_known l1 x y in
let l' = map_known interleave l in  ...
```

Closure conversion runs next, after which all function applications will fetch the code and the environment from the closure pair. We use the constructor C_{cc} for closure pairs and the constructor C_{f} for the closure environment of the function `interleave`. The projections from statically known closure-pairs will be statically evaluated by shrinking, and the constructed closure will be dead code, and therefore deleted. The redundant environment argument of known functions `interleave_known` and `map_known` will be deleted by dead parameter elimination. The only function that will get a heap-allocated environment and closure pair is the escaping `interleave` function. The final code is shown below.

```
fun interleave_known l x y =
  case l of
  | [] => []
  | z :: l => x::y::z::(interleave_known l x y)
  end  in
fun interleave l env = let x = env.1 in let y = env.2 in interleave_known l x y  in
let env = Cf(x,y) in let interleave_clo = Ccc(interleave,env) in
let l1 = [1, 2, 3] in let l = [l1, l1, l1] in
let l1' = interleave_known l1 x y in
let l' = map_known interleave_clo l in ...
```

4 SEMANTICS AND COMPILER CORRECTNESS

Our goal is to prove that our λ_{ANF} pipeline compiles programs correctly, that is, the target program preserves the source program's observable behaviors, such as termination and divergence. We will prove compiler correctness both for closed, whole programs and open programs, compiled separately and linked at the target level. A program obtained after linking modules at the target level should preserve the behavior of the program obtained by linking the modules at the source-level. We are interested not only in compiling programs separately, but also in compiling them with *different* optimizations enabled. Not every optimization will be beneficial for every program so we might turn optimizations off. We might also improve some λ_{ANF} optimizations (*e.g.*, implement more efficient closure environment representations [Shao and Appel 2000]) or add new λ_{ANF} optimizations. Then we should be able to link newly compiled modules with previously compiled modules.

For simplicity, we omit mutually defined functions from the formal definitions the rest of the paper. Our implementation and mechanization fully support mutually recursive functions.

4.1 Semantics

To prove correctness of a compiler we formalize the semantics of the source language (λ_{ANF}) and target language (also λ_{ANF}).[4] The semantics of λ_{ANF} is a fuel-based, big-step semantics. It relates a term with a final result under a fuel value; the final result will be an out-of-time exception if there is insufficient fuel to carry out a computation. Using this fuel-based definition we can semantically characterize nonterminating terms and hence state and prove that divergence is preserved through our compiler.

We first define λ_{ANF} values, evaluation environments, and results (fig. 2). A λ_{ANF} value is either a constructed value or a closure value consisting of an environment and a function definition. An environment is a partial map from variables to values. The final result returned by the semantics is either a value (wrapped in a Res constructor) or an out-of-time exception (OOT).

$$\begin{array}{rclr}
v \in \mathtt{Val} & ::= & \mathtt{C}(\overline{v}) \mid \langle \sigma, \mathtt{fun}\ f\ \overline{x} = e \rangle & \text{Values} \\
\sigma \in \mathtt{Env} & = & \mathtt{Var} \rightharpoonup \mathtt{Val} & \text{Environments} \\
r \in \mathtt{Res} & ::= & \mathsf{Res}(v) \mid \mathsf{OOT} & \text{Results}
\end{array}$$

Fig. 2. λ_{ANF} values, environments, and results.

Our semantic definition is indexed with a fuel and a trace value. The fuel keeps track of the length of the derivation tree of the evaluation. It is used to define diverging executions and show that diverging behaviors are preserved. The trace value counts the number of application steps (let-bound or tail) in the derivation tree. We found that this is necessary in order to express an invariant for the inlining transformation in ANF form. In section 6, we give an upper bound of the fuel consumption of a program before inlining as a function of both the fuel consumption of the inlinlined program and the trace of the input program. Our mechanized semantic definitions are parameterized by two abstract commutative monoids that can be instantiated with different fuel and trace models. But since in this paper we use one concrete fuel and trace instantiation, we only present those.

We now have all the tools we need to define the semantics. The judgment $(\sigma, e)\ {}^{c}\Downarrow^{t}\ r$ asserts that the configuration consisting of environment σ and term e evaluates with fuel c to result r incurring a trace t. Recall that a result is either a value or an out of time exception. The semantic judgment is mutually defined with the auxiliary judgment $(\sigma, e)\ {}^{c}\!\downarrow^{t}\ r$. Conceptually, $\Downarrow$ is responsible for the fuel and trace profiling, whereas $\downarrow$ evaluates the outermost constructor of the term.

Each evaluation step first goes through the rule Step that does the bookkeeping of trace and fuel resources and invokes the $\downarrow$ judgment to evaluate the outermost constructor of the term. The fuel needed to evaluate the outermost constructor of a term e is always one unit (given by $\mathsf{fuel}(e)$). The

[4]Our target program is in the $\lambda^{\mathsf{C}}_{\mathsf{ANF}}$ subset. Any $\lambda^{\mathsf{C}}_{\mathsf{ANF}}$ program can be interpreted using λ_{ANF} semantics or using a simpler $\lambda^{\mathsf{C}}_{\mathsf{ANF}}$ semantics, which does not use closure values, and the observable result will be the same. We move to closure-less semantics during our code generation proof, so we don't need $\lambda^{\mathsf{C}}_{\mathsf{ANF}}$ semantics for the results covered in this paper.

$$\frac{\sigma(\overline{y}) = \overline{v} \qquad (\sigma[x \mapsto \mathtt{C}(\overline{v})], e)\ {}^{c}{\Downarrow}^{t}\ r}{(\sigma, \mathtt{let}\ x = \mathtt{C}(\overline{y})\ \mathtt{in}\ e)\ {}^{c}{\downarrow}^{t}\ r}\ \textsc{Constr} \qquad \frac{\sigma(x) = \mathtt{C}_i(\overline{v}) \qquad (\sigma, e_i)\ {}^{c}{\Downarrow}^{t}\ r}{(\sigma, \mathtt{case}\ x\ \mathtt{of}\ \{\mathtt{C}_i\ \rightarrow\ e\}_{i \in I})\ {}^{c}{\downarrow}^{t}\ r}\ \textsc{Case}$$

$$\frac{\begin{array}{c}\sigma(y) = \mathtt{C}(v_1, \ldots, v_j, \ldots, v_n)\\ (\sigma[x \mapsto v_j], e)\ {}^{c}{\Downarrow}^{t}\ r\end{array}}{(\sigma, \mathtt{let}\ x = y.j\ \mathtt{in}\ e)\ {}^{c}{\downarrow}^{t}\ r}\ \textsc{Proj} \qquad \frac{(\sigma[f \mapsto \langle \sigma, \mathtt{fun}\ f\ \overline{x} = e_1 \rangle], e_2)\ {}^{c}{\Downarrow}^{t}\ r}{(\sigma, \mathtt{fun}\ f\ \overline{x} = e_1\ \mathtt{in}\ e_2)\ {}^{c}{\downarrow}^{t}\ r}\ \textsc{Fun}$$

$$\frac{\begin{array}{c}\sigma(f) = \langle \sigma_g, \mathtt{fun}\ g\ \overline{z} = e_g \rangle \qquad |\overline{y}| = |\overline{z}| \qquad \sigma(\overline{y}) = \overline{v}\\ (\sigma_g[\overline{z} \mapsto \overline{v}][g \mapsto \langle \sigma_g, \mathtt{fun}\ g\ \overline{z} = e_g \rangle], e_g)\ {}^{c_1}{\Downarrow}^{t_1}\ \mathsf{Res}(v_1) \qquad (\sigma[x \mapsto v_1], e)\ {}^{c_2}{\Downarrow}^{t_2}\ r\end{array}}{(\sigma, \mathtt{let}\ x = f\ \overline{y}\ \mathtt{in}\ e)\ {}^{c_1+c_2}{\downarrow}^{t_1+t_2}\ r}\ \textsc{Let-app}$$

$$\frac{\begin{array}{c}\sigma(f) = \langle \sigma_g, \mathtt{fun}\ g\ \overline{z} = e_g \rangle\\ |\overline{y}| = |\overline{z}| \qquad \sigma(\overline{y}) = \overline{v} \qquad (\sigma_g[\overline{z} \mapsto \overline{v}][g \mapsto \langle \sigma_g, \mathtt{fun}\ g\ \overline{z} = e_g \rangle], e_g)\ {}^{c}{\Downarrow}^{t}\ \mathsf{OOT}\end{array}}{(\sigma, \mathtt{let}\ x = f\ \overline{y}\ \mathtt{in}\ e)\ {}^{c}{\downarrow}^{t}\ \mathsf{OOT}}\ \textsc{Let-app-oot}$$

$$\frac{\begin{array}{c}\sigma(f) = \langle \sigma_g, \mathtt{fun}\ g\ \overline{z} = e_g \rangle\\ |\overline{y}| = |\overline{z}| \qquad \sigma(\overline{y}) = \overline{v} \qquad (\sigma_g[\overline{z} \mapsto \overline{v}][g \mapsto \langle \sigma_g, \mathtt{fun}\ g\ \overline{z} = e_g \rangle], e_g)\ {}^{c}{\Downarrow}^{t}\ r\end{array}}{(\sigma, f\ \overline{y})\ {}^{c}{\downarrow}^{t}\ r}\ \textsc{App}$$

$$\frac{\sigma(x) = v}{(\sigma, \mathtt{ret}(x))\ {}^{0}{\downarrow}^{0}\ \mathsf{Res}(v)}\ \textsc{Ret} \qquad \frac{i < \mathsf{fuel}(e)}{(\sigma, e)\ {}^{i}{\Downarrow}^{0}\ \mathsf{OOT}}\ \mathsf{OOT} \qquad \frac{(\sigma, e)\ {}^{c}{\downarrow}^{t}\ r}{(\sigma, e)\ {}^{c+\mathsf{fuel}(e)}{\Downarrow}^{t+\mathsf{trace}(e)}\ r}\ \textsc{Step}$$

$$\text{where} \quad \mathsf{fuel}(e) = 1 \qquad \begin{aligned} \mathsf{trace}(f\ \overline{y}) &= 1\\ \mathsf{trace}(\mathtt{let}\ x = f\ \overline{y}\ \mathtt{in}\ e) &= 1\\ \mathsf{trace}(e) &= 0 \quad \text{if } e \text{ is not a function call}\end{aligned}$$

Fig. 3. The semantics of λ_{ANF}.

trace is unit only when the expression is an application and zero otherwise (given by $\mathsf{trace}(e)$). The rules of $\downarrow$ are straightforward. Observe that the semantics of λ_{ANF} does not permit partial application of functions. All functions must be fully applied to their arguments. This trivially holds after ANF translation since functions are curried and all functions and applications are unary. Multi-argument functions are introduced by the λ_{ANF} pipeline and are never partially applied. The rule OOT throws an out-of-time exception whenever there is not enough fuel to evaluate the outermost constructor of the term. In this case, an empty trace is recorded.

Using this semantics, we can define what it means for a term to diverge. A term diverges, written $(\sigma, e) \Uparrow$, if for all fuel values, the computation runs out of time.

$$(\sigma, e) \Uparrow \quad \stackrel{\text{def}}{=} \quad \forall c, \exists t,\ (\sigma, e)\ {}^{c}{\Downarrow}^{t}\ \mathsf{OOT}$$

When e is closed and evaluates in the empty environment we will write $e\ {}^{c}{\Downarrow}^{t}\ r$ and $e \Uparrow$.

4.2 Correctness Specification

With the semantics in hand, we can state the top-level correctness specification for the λ_{ANF} pipeline. First, we state program refinement for whole program compilation. If a closed source program terminates yielding a final value, then the target term must terminate yielding a value that refines the source value. If a source program diverges, then so must the target program. This is captured by a *behavioral refinement* specification $e_{\text{src}} \supseteq_{\mathsf{B}} e_{\text{trg}}$.

$$\begin{aligned} e_{\text{src}} \supseteq_{\mathsf{B}} e_{\text{trg}} \quad \overset{\text{def}}{=} \quad & (e_{\text{src}} \Downarrow \mathsf{Res}(v_{\text{src}}) \;\Rightarrow\; \exists\, v_{\text{trg}},\; e_{\text{trg}} \Downarrow \mathsf{Res}(v_{\text{trg}}) \;\wedge\; v_{\text{src}} \supseteq_{\mathsf{V}} v_{\text{trg}}) \;\wedge \\ & e_{\text{src}} \Uparrow \;\Rightarrow\; e_{\text{trg}} \Uparrow \end{aligned}$$

We say that a compiler comp is correct for whole programs if for all terms e, $e \supseteq_{\mathsf{B}} \mathsf{comp}(e)$. The value refinement relation $\supseteq_{\mathsf{V}}$ specifies when a target value refines a source value:

$$\mathsf{C}_1(v_1, \ldots, v_m) \supseteq_{\mathsf{V}} \mathsf{C}_2(v'_1, \ldots, v'_n) \overset{\text{def}}{=} \mathsf{C}_1 = \mathsf{C}_2 \;\wedge\; n = m \;\wedge\; \forall i,\; v_i \supseteq_{\mathsf{V}} v'_i$$

$$\langle \sigma, \mathtt{fun}\; f\; \overline{x} = e \rangle \supseteq_{\mathsf{V}} \mathsf{C}_{\text{cc}}(v_f, e_f) \overset{\text{def}}{=} \mathtt{True}$$

The value refinement relates constructed values that have the same outermost constructor and pairwise related values. It relates λ_{ANF} closure values with explicitly constructed closure values (with constructor tag C_{cc}), but it does not specify anything about the behavior of the function parts of the closure. Why? We use behavioral refinement to relate closed, top-level programs. When we run a whole, top-level program we can observe only the first-order components of the result. First order results are constructed values that can be inspected with appropriate knowledge of the memory representation. In contrast, to observe a function value, a program must be *linked* with another program that will apply the result to some arguments. We address this next.

First, we need to define what linking programs means in the λ_{ANF} language. Our linking operation links a client program e_{client} that has a free variable x with a library program e_{lib} that computes the value of the variable x. It is defined below.

$$[x \mapsto e_{\text{lib}}]e_{\text{client}} \overset{\text{def}}{=} \mathtt{fun}\; f\; [] = e_{\text{lib}}\; \mathtt{in}\; \mathtt{let}\; x = f\; []\; \mathtt{in}\; e_{\text{client}}$$

Intuitively, we can think of the linking operator as a closing substitution that substitutes the variable x in the client program with the term e_{lib}. But the syntactic restrictions of A-normal form mean that in λ_{ANF} we cannot simply substitute an identifier for an expression or write $\mathtt{let}\; x = e_{\text{lib}}\; \mathtt{in}\; e_{\text{client}}$. Therefore we use a zero-arity function to wrap the computation e_{lib} and we bind the result of application of this function to an empty list of arguments to the variable x. The linking operator can be generalized to multiple external references.

Using behavioral refinement and the linking operator we can define what it means for separate compilation to be correct. Two programs e and e' that are separately compiled with compilers comp_1 and comp_2 *may be safely linked* if we have:

$$[x \mapsto e']e \supseteq_{\mathsf{B}} [x \mapsto \mathsf{comp}_2(\mathsf{e}')]\mathsf{comp}_1(\mathsf{e})$$

The specification for separate compilation asserts that the whole target program obtained by linking the two compiled programs refines the behavior of the whole source program obtained by linking the two programs at the source level.

When the same compiler is used to compile both programs (*i.e.*, $\mathsf{comp}_1 = \mathsf{comp}_2$) and compilation commutes with linking (*i.e.*, $\mathsf{comp}_1([x \mapsto e'_s]e_s) = [x \mapsto \mathsf{comp}_1(e'_s)]\mathsf{comp}_1(e_s)$), it is trivial to prove behavioral refinement for linking. However, neither is true for the λ_{ANF} pipeline. We want to be able to use different optimizations when compiling programs separately. Even if we were willing to restrict ourselves to the same pipeline, the commutation requirement does not hold for multiple reasons (*e.g.*, alpha-conversion, possible inlining of the zero-arity linking function f). So there is no easy recipe to show correctness for separate compilation. The novel compositional compiler correctness technique that we introduce in section 5 allows us to show behavioral refinement for separate compilation.

4.3 Divergence Preservation

We now formalize the statement that we outlined in section 2.2: in order to show that divergence is preserved it suffices to show that the fuel consumption of the source program is upper bounded by a strictly monotonic function of the fuel consumption of the target program.

First, we need to prove a fuel-monotonicity property for our semantics: if a program times out for some fuel value c, then it times out for all smaller fuel values.

Lemma 4.1 (Fuel monotonicity). *Assume that* $(\sigma, e)\ {}^{c}{\Downarrow}^{t}\ \mathsf{OOT}$. *Then, for any* $c' \leq c$, *there exist* $t' \leq t$ *such that* $(\sigma, e)\ {}^{c'}{\Downarrow}^{t'}\ \mathsf{OOT}$.

Using the above lemma, we can prove the desired divergence preservation lemma.

Lemma 4.2 (Divergence preservation). *Let* f *be a function* $\mathbb{N} \to \mathbb{N}$ *such that* $f(x) \leq f(y) \Rightarrow x \leq y$. *Assume that for two configurations* (σ_1, e_1) *and* (σ_2, e_2) *we know that if* $(\sigma_1, e_1)\ {}^{c_1}{\Downarrow}^{t_1}\ \mathsf{OOT}$ *then there exist* c_2 *and* t_2 *such that* $(\sigma_2, e_2)\ {}^{c_2}{\Downarrow}^{t_2}\ \mathsf{OOT}$ *and* $c_1 \leq f(c_2)$. *Then if* $(\sigma_1, e_1) \Uparrow$ *we have that* $(\sigma_2, e_2) \Uparrow$.

Proof. Let c be a fuel value. We must show that there exists t such that $(\sigma_2, e_2)\ {}^{c}{\Downarrow}^{t}\ \mathsf{OOT}$. From the hypothesis that e_1 is a diverging program, we know that $(\sigma_1, e_1)\ {}^{f(c)}{\Downarrow}^{t_1}\ \mathsf{OOT}$ for some t_1. Therefore, we can derive that $(\sigma_2, e_2)\ {}^{c_2}{\Downarrow}^{t_2}\ \mathsf{OOT}$ for some c_2 and t_2 such that $f(c) \leq f(c_2)$. But from the hypothesis about f we have that $c \leq c_2$. From the fuel monotonicity property of our semantics we obtain $t \leq t_2$ such that $(\sigma_2, e_2)\ {}^{c}{\Downarrow}^{t}\ \mathsf{OOT}$. □

In the next section we will set up a logical relation that let us impose arbitrary binary relations on the fuel consumption of two programs. If such relation implies the required bound of lemma 4.2, then we can establish divergence preservation. For transformations that do not reduce fuel we take f to be the identity function. In order express to express an upper bound the fuel consumption of the program before inlining in terms of the fuel consumption of the program after inlining, we will use the number of function applications performed during the source program's execution, which is captured by the trace value of the semantics of λ_{ANF}.

5 COMPOSITIONAL PROOF FRAMEWORK

To prove correctness of λ_{ANF} transformations, we use untyped, step-indexed logical relations. Although logical relations are more commonly indexed by types, untyped logical relations also appear in the literature [Acar et al. 2008; Paraskevopoulou and Appel 2019; Owens et al. 2017; Georges et al. 2021]. We set up two different logical relations: a reflexive relation to prove correct transformations that don't change the representation of function values, and a nonreflexive relation to prove correct closure conversion, which does change the representation of functions. Then, as we outlined in section 2, we compose these relations to derive a top-level relation that is both adequate and compatible with linking. Ultimately, that will allow us to derive a separate compilation theorem for our pipeline that asserts that we can safely link programs complied through the λ_{ANF} pipeline, perhaps using different λ_{ANF} optimizations.

The reflexive logical relation is shown in fig. 4. It consists of a value relation $\boldsymbol{\mathcal{V}}$, a result relation $\boldsymbol{\mathcal{R}}$, an expression relation $\boldsymbol{\mathcal{E}}$, and an environment relation $\boldsymbol{\mathcal{C}}$.

The expression relation is $\boldsymbol{\mathcal{E}}^k((\sigma_1, e_1), (\sigma_2, e_2))\ \{Q_L; Q_G\}$. The first argument k is the usual step index that is needed for the well-foundedness of the definition. The next two arguments are the configurations (pairs of environments and expressions) that are being related. The last two arguments are two relational invariants: a *local invariant* and a *global invariant*. This is the only nonstandard feature of our logical relations. These invariants are binary relations over pairs of a fuel and trace value. The relational invariants allow us to establish the bound that we need to show divergence preservation (lemma 4.2). This bound will typically be different for each λ_{ANF} transformation. The local invariant is imposed on the fuel and trace values of the executions of the current configurations while the global invariant is imposed on the future execution of functions in the results through the value relation. Local and global invariants need to be separate because global invariants are used to relate only whole-function executions, while local invariants relate programs at arbitrary points during execution. Hence, they might need to be instantiated with a different relation; we give an example of this later in this section.

The result relation $\boldsymbol{\mathcal{R}}$ relates two results that are either out-of-time exceptions or related values. The value relation $\boldsymbol{\mathcal{V}}$ relates λ_{ANF} values. Two constructed values are related if they are constructed with the same constructor and their constructor arguments have the same lengths and are pairwise related (written $\boldsymbol{\mathcal{V}}^k(\overline{v_1}, \overline{v_2})\ \{Q\}$). Two closure values are related if for any two lists of parameters related at some strictly smaller step index i, the two configurations that consist of the body of the function part of the (corresponding) closure and the environment part of the closure (extended with appropriate bindings) are related. The closure environments are extended with bindings that map the formal parameters to the actual parameters and the function name to its closure value, which is needed for recursive functions. The value relation is indexed only with a global invariant that relates the fuel and trace consumption of related function bodies. When the expression relation is invoked from the value relation for closures, the global invariant is used to instantiate both the local and the global invariant of the expression relation.

The environment relation $\boldsymbol{\mathcal{C}}$ lifts the value relation to a subdomain S of two environments and it is defined using an auxiliary variable relation $\boldsymbol{\mathcal{X}}$. In particular, for any variable in the set S, we assert that if it is defined in the domain of the first environment then it is also defined in the domain of the second environment and their values are related with the value relation. The environment relation is used to assert that the environment parts of two configurations are logically related. It is convenient to restrict this relation to a variable subset S that is relevant for a program's execution (which is typically the set of free variables of a term).

The closure-conversion relation is shown in fig. 5. Only the closure case of the value relation $\boldsymbol{\mathcal{V}}_{\mathsf{CC}}$ is different than the reflexive relation; therefore, we show only show this. The other cases of $\boldsymbol{\mathcal{V}}_{\mathsf{CC}}$ and the rest of the definitions ($\boldsymbol{\mathcal{E}}_{\mathsf{CC}}$, $\boldsymbol{\mathcal{C}}_{\mathsf{CC}}$, and $\boldsymbol{\mathcal{R}}_{\mathsf{CC}}$) are defined as before. The $\boldsymbol{\mathcal{V}}_{\mathsf{CC}}$ relation for closures relates a λ_{ANF} closure value with a *constructed* closure value. The first component for the constructed value is a closure value (*i.e.*, the code component of the closure) and the second component an environment (which is also a constructed value). It might seem odd that the code component of the constructed closure is itself a (primitive, not constructed) closure. However, after closure conversion, functions do not become immediately closed. They might still contain free variables that are references to other (known, not escaping) functions. Functions will become closed when they are all moved to a mutually defined function block after the hoisting transformation. We formally prove that after hoisting all functions are hoisted to the top-level and are *closed*. The compiler moves to a semantics without closure values but just bare functions during the code-generation phase proof. Technically speaking, we could transition to a representation without closures within this framework by using a third logical relation that moves to a closure-less semantics and function representation.

Correctness. We show correctness of transformations by showing that under any logically related environments the source and target terms are related for an appropriate choice of local and

Value relation

$$\mathcal{V}^k(\mathsf{C}_1(\overline{v_1}), \mathsf{C}_2(\overline{v_2}))\ \{Q\} \overset{\text{def}}{=} \mathsf{C}_1 = \mathsf{C}_2\ \wedge\ \mathcal{V}^k(\overline{v_1}, \overline{v_2})\ \{Q\}$$

$$\mathcal{V}^k(\langle \sigma_1, \mathsf{fun}\ f\ \overline{x} = e_1\rangle, \langle \sigma_2, \mathsf{fun}\ g\ \overline{y} = e_2\rangle)\ \{Q\} \overset{\text{def}}{=}$$
$$\forall\ i < k\ \overline{v_1}\ \overline{v_2},\ \mathcal{V}^i(\overline{v_1}, \overline{v_2})\ \{Q\}\ \Rightarrow\ |\overline{x}| = |\overline{v_1}|\ \Rightarrow\ |\overline{y}| = |\overline{v_2}|\ \wedge\ \mathcal{E}^i((\sigma_1', e_1), (\sigma_2', e_2))\ \{Q; Q\}$$

Where $\sigma_1' = \sigma_1[\overline{x} \mapsto \overline{v_1}, f \mapsto \langle \sigma_1, \mathsf{fun}\ f\ \overline{x} = e_1\rangle]$
and $\sigma_2' = \sigma_2[\overline{y} \mapsto \overline{v_2}, g \mapsto \langle \sigma_2, \mathsf{fun}\ g\ \overline{y} = e_2\rangle]$.

$$\mathcal{V}^k(v_1, v_2)\ \{Q\} \overset{\text{def}}{=} \mathtt{False} \qquad \text{Otherwise.}$$

Result relation

$$\mathcal{R}^k(\mathsf{OOT}, \mathsf{OOT})\ \{Q\} \overset{\text{def}}{=} \mathtt{True}$$
$$\mathcal{R}^k(\mathsf{Res}(v_1), \mathsf{Res}(v_2))\ \{Q\} \overset{\text{def}}{=} \mathcal{V}^k(v_1, v_2)\ \{Q\}$$
$$\mathcal{R}^k(r_1, r_2)\ \{Q\} \overset{\text{def}}{=} \mathtt{False} \qquad \text{Otherwise.}$$

Expression relation

$$\mathcal{E}^k((\sigma_1, e_1), (\sigma_2, e_2))\ \{Q_L; Q_G\} \overset{\text{def}}{=}$$
$$\forall c_1\ r_1\ t_1,\ c_1 \leq k\ \Rightarrow\ (\sigma_1, e_1)\ {}^{c_1}\Downarrow^{t_1}\ r_1\ \Rightarrow$$
$$\exists\ c_2\ r_2\ t_2,\ (\sigma_2, e_2)\ {}^{c_2}\Downarrow^{t_2}\ r_2\ \wedge\ Q_L(c_1, t_1)\ (c_2, t_2)\ \wedge \mathcal{R}^{k-c_1}(r_1, r_2)\ \{Q_G\}$$

Variable relation

$$\mathcal{X}^k(\sigma_1, x)\ (\sigma_2, y)\ \{Q\} \overset{\text{def}}{=} \sigma_1(x) = v_1\ \Rightarrow\ \exists v_2, \sigma_2(y) = v_2\ \wedge\ \mathcal{V}^k(v_1, v_2)\ \{Q\}$$

Environment relation

$$S \vdash \mathcal{C}^k(\sigma_1, \sigma_2)\ \{Q\} \overset{\text{def}}{=} \forall\ (x \in S),\ \mathcal{X}^k(\sigma_1, x)\ (\sigma_2, x)\ \{Q\}$$

Fig. 4. The reflexive logical relation.

$$\mathcal{V}^k_{\mathsf{CC}}(\langle \sigma_1, \mathsf{fun}\ f\ \overline{x} = e_1\rangle, \mathsf{C}_{\mathsf{CC}}(\langle \sigma_2, \mathsf{fun}\ g\ \gamma :: \overline{y} = e_2\rangle, e))\ \{Q\} \overset{\text{def}}{=}$$
$$\forall\ i < k\ \overline{v_1}\ \overline{v_2},\ \mathcal{V}^i_{\mathsf{CC}}(\overline{v_1}, \overline{v_2})\ \{Q\}\ \Rightarrow\ |\overline{x}| = |\overline{v_1}|\ \Rightarrow\ |\overline{y}| = |\overline{v_2}|\ \wedge\ \mathcal{E}^i_{\mathsf{CC}}((\sigma_1', e_1), (\sigma_2', e_2))\ \{Q; Q\}$$

Where $\sigma_1' = \sigma_1[\overline{x} \mapsto \overline{v_1}, f \mapsto \langle \sigma_1, \mathsf{fun}\ f\ \overline{x} = e_1\rangle]$
and $\sigma_2' = \sigma_2[\gamma \mapsto e, \overline{y} \mapsto \overline{v_2}, f \mapsto \langle \sigma_2, \mathsf{fun}\ g\ \overline{y} = e_2\rangle]$.

Fig. 5. The closure conversion logical relation (excerpt).

global invariants. In particular, we define a top-level logical relation:

$$\overline{\mathcal{E}}\ (e_1, e_2)\ \{Q_L; Q_G\} \overset{\text{def}}{=}\ \ \forall\ k\ \sigma_1\ \sigma_2,\ \mathsf{fv}(e_1) \vdash \mathcal{C}^k(\sigma_1, \sigma_2)\ \{Q_G\}\ \Rightarrow\ \mathsf{fv}(e) \subseteq \mathsf{dom}(\sigma_1)\ \Rightarrow\ \mathcal{E}^k((e_1, \sigma_1), (e_2, \sigma_2))\ \{Q_L; Q_G\}$$

Similarly, we define $\overline{\mathcal{E}}_{\mathsf{CC}}\ (e_1, e_2)\ \{Q_L; Q_G\}$ for the $\mathcal{E}_{\mathsf{CC}}$ relation. Then for each transformation, trans, we show that $\overline{\mathcal{E}}\ (e, \mathsf{trans}(e))\ \{Q_L; Q_G\}$ (or in the case of closure conversion $\overline{\mathcal{E}}_{\mathsf{CC}}\ (e, \mathsf{trans}(e))\ \{Q_L; Q_G\}$) for appropriate Q_L and Q_G.

Notice that we require that the free variables of the source term must be defined in the source environment—this is required by the proofs of certain transformations (lambda lifting and closure conversion). In both transformations, the target term accesses free variables earlier than the source term, and we need this additional assumption so that target terms do not get stuck earlier than source terms.

Local* vs. *global invariants. Typically, when we consider top-level programs the local and the global invariants are the same. But during a proof, a local invariant may become (locally) different from the global invariant as we consider intermediate states of the execution of the source and target programs. We illustrate this with an example. Let ${}^{k}{=}$ be a relation that relates two pairs (c_1, t_1) and (c_2, t_2) of a fuel and trace value if and only if $c_1 + k = c_2$ (we ignore the trace value in this example for simplicity). The logical relation (as we will formally state later on) is reflexive. Hence, we can establish that:

$$\mathcal{E}^k((\sigma, \mathtt{fun}\ f\ x = e'\ \mathtt{in}\ e), (\sigma, \mathtt{fun}\ f\ x = e'\ \mathtt{in}\ e))\ \{\ {}^{0}{=};\ {}^{0}{=}\}$$

Unsurprisingly, the fuel consumption is the same for the left and right configurations since it is exactly the same program.

Now assume that we wish to relate the configuration $(\sigma[f \mapsto \langle \sigma, \mathtt{fun}\ f\ \overline{x} = e' \rangle], e)$ with the configuration $(\sigma, \mathtt{fun}\ f\ x = e'\ \mathtt{in}\ e)$. These configurations are clearly related: they compute exactly the same result. The only difference is that in the first configuration, we have performed execution step. The global invariant should still be ${}^{0}{=}$ since the results are identical and therefore any two related functions in the results will still have the same fuel consumption. However, the local invariant is not the same anymore. If the first configuration consumes fuel f then the second configuration consumes fuel $f + 1$ since it has to execute one more step. Therefore we have:

$$\mathcal{E}^k((\sigma[f \mapsto \langle \sigma, \mathtt{fun}\ f\ \overline{x} = e' \rangle], e), (\sigma, \mathtt{fun}\ f\ x = e'\ \mathtt{in}\ e))\ \{\ {}^{1}{=};\ {}^{0}{=}\}$$

The above two programs are related with a different local and global invariant. Therefore, keeping the local and the global invariants separate gives us the flexibility to relate programs at points of execution where the global invariant does not hold.

Compatibility Lemmas. As usual, we formally prove compatibility rules for the logical relation, asserting that the logical relation is closed under language constructions. These lemmas are used to reason compositionally about program relatedness. Because our relation is indexed with local and global invariants, to prove the compatibility lemmas we need to assume that the invariants satisfy some compatibility properties too. These help us establish the invariants compositionally, by asserting that the invariants are preserved when a step of computation happens. As an example, we consider the compatibility rule for constructors. The lemma states that two constructors are related if their arguments are pairwise related in the environments, and their continuations are related in the environments extended with related bindings.

Lemma 5.1 (Compatibility (constructor)). *Assume that* $Q_L\ (0,0)\ (0,0)$ *and that* $Q_L\ (c_1, t_1)\ (c_2, t_2) \Rightarrow Q_L\ (c_1 + 1, t_1)\ (c_2 + 1, t_2)$. *If*

- $\mathcal{X}^k(\sigma_1, \overline{y_1})\ (\sigma_2, \overline{y_2})\ \{Q_G\}$
- $\forall\ \overline{v_1}\ \overline{v_2},\ \mathcal{V}^k(\overline{v_1}, \overline{v_2})\ \{Q_G\}\ \Rightarrow\ \mathcal{E}^k((\sigma_1[x_1 \mapsto \mathtt{C}(\overline{v_1})], e_1), (\sigma_2[x_2 \mapsto \mathtt{C}(\overline{v_2})], e_2))\ \{Q_L; Q_G\}$

then $\mathcal{E}^k((\sigma_1, \mathtt{let}\ x_1 = \mathtt{C}(\overline{y_1})\ \mathtt{in}\ e_1), (\sigma_2, \mathtt{let}\ x_2 = \mathtt{C}(\overline{y_2})\ \mathtt{in}\ e_2))\ \{Q_L; Q_G\}$.

The above lemma requires that the local invariant holds for zero fuel and trace value. This is useful to establish the local invariant when both configurations time out. It also assumes that the local invariant is preserved when adding the fuel needed to evaluate a constructor to two related fuel values. This is needed to establish the local invariant. Assume that c_1 and t_1 (resp. c_2 and t_2) is the fuel and trace of expression e_1 (resp. e_2). From the hypothesis that e_1 and e_2 are related, we know that $Q_L\ (c_1, t_1)\ (c_2, t_2)$. Using the compatibility assumption for the local invariant, we derive that Q_L $(c_1 + 1, t_1)$ $(c_2 + 1, t_2)$, which relates the fuel and trace values of $\texttt{let}\ x_1 = \texttt{C}(\overline{y_1})\ \texttt{in}\ e_1$ and $\texttt{let}\ x_2 = \texttt{C}(\overline{y_2})\ \texttt{in}\ e_2$.

Similarly, we need compatibility rules for resource invariants to show the rest of the compatibility lemmas of the logical relation. We define the predicate Compat Q_G Q_L to mean that the local invariant Q_L and global invariant Q_G satisfy these compatibility rules. It asserts that:

(1) Q_L $(0, 0)$ $(0, 0)$. This is required when both configurations time out.
(2) Q_L $(1, 0)$ $(1, 0)$. This is required when both configurations return. Recall from the definition of the semantics that return does not incur any trace.
(3) Q_L (c_1, t_1) $(c_2, t_2) \Rightarrow Q_L$ $(c_1 + 1, t_1)$ $(c_2 + 1, t_2)$.
This is required for the inductive constructors of the language that are not function applications, as illustrated in lemma 5.1.
(4) Q_G (c_1, t_1) $(c_2, t_2) \Rightarrow Q_L$ (c_3, t_3) $(c_4, t_4) \Rightarrow Q_L$ $(c_1 + c_3 + 1, t_1 + t_3 + 1)$ $(c_2 + c_4 + 1, t_2 + t_4 + 1)$.
This is required for the compatibility lemma of let-bound application. The first premise holds for the fuel and trace values of the function bodies, which are related by the global invariant. The second premise holds for the fuel and trace values for the expressions that are evaluated after the function calls (*i.e.*, the expressions that the calls are let-bound in). The result adds the fuel and trace values of the function calls and their continuations and adds a unit to both the fuel and the trace values (since applications require a unit of fuel and a unit of trace).
(5) Q_G (c_1, t_1) $(c_2, t_2) \Rightarrow Q_L$ $(c_1 + 1, t_1 + 1)$ $(c_2 + 1, t_2 + 1)$.
This is required for the compatibility lemma of tail-call applications.

Reflexivity. The $\mathcal{E}$ relation is reflexive. We take advantage of this property later on, in order to prove a linking theorem for pipelines that use an arbitrary number for same-representation transformations. Reflexivity is proved using the compatibility lemmas for the logical relation; hence, it assumes that the invariants satisfy the compatibility rules of the previous paragraph.

Lemma 5.2 (Reflexivity).
Assume that Compat Q_G Q. *If* $\text{fv}(e) \vdash \mathcal{C}^k(\sigma_1, \sigma_2)\ \{Q\}$ *then* $\mathcal{E}^k((\sigma_1, e), (\sigma_2, e))\ \{Q; Q_G\}$.

Transitivity. The $\mathcal{E}$ relation can be transitively composed:

Lemma 5.3 (Transitivity). *Assume that* Compat Q_G Q_G *and that* $Q_G \circ Q_G \subseteq Q_G$. *If* $\mathcal{E}^k((\sigma_1, e_1), (\sigma_2, e_2))\ \{Q_1; Q_G\}$ *and* $\forall\ k,\ \mathcal{E}^k((\sigma_2, e_2), (\sigma_3, e_3))\ \{Q_2; Q_G\}$ *then* $\mathcal{E}^k((\sigma_1, e_1), (\sigma_3, e_3))\ \{Q_1 \circ Q_2; Q_G\}$.

This lemma requires that the global invariant is semi-idempotent: $(Q_G \circ Q_G \subseteq Q_G)$. This is necessary in order to transitively compose the value relation when the results are closures (using the induction hypothesis—the induction is on the step-index). We use this property in the correctness proof of the uncurrying transformation, which is expressed as the transitive closure of a rewrite step. For the simple invariant of this transformation the semi-idempotency requirement holds. However, the relation $\mathcal{E}$ is not transitive in the general case. We cannot compose the proofs of transformations that have different global invariants or have global invariants that are not idempotent.

Adequacy and Compatibility with Linking are two important properties of the logical relations. A relation is adequate if it implies the behavioral refinement that we wish to establish. To prove adequacy we need to establish divergence preservation. Therefore, we need to know that the local invariant implies the upper bound of lemma 4.2. This is captured by the following definition.

Definition 5.4 (Invariant, upper bound). UpperBound Q holds iff there exists a function f such that $\forall\ x\ y,\ f(x) \leq f(y) \Rightarrow x \leq y$ and for all fuel values c_1, c_2 and trace values t_1, t_2 if $Q(c_1, t_1)$ (c_2, t_2) then $c_1 \leq f(c_2)$.

Using the UpperBound predicate we can state and prove adequacy for the logical relations. The logical relation $\mathcal{E}_{\mathrm{CC}}$ implies the behavioral refinement of the previous section. This is stated below ($\varnothing$ is the empty environment).

Lemma 5.5 (Adequacy). *Assume* UpperBound Q_L. *Then,*

$$(\forall\, k,\ \mathcal{E}^k_{\mathrm{CC}}((\varnothing, e_1), (\varnothing, e_2))\ \{Q_L; Q_G\})\ \Rightarrow\ e_1 \supseteq_{\mathrm{B}} e_2$$

The logical relation $\mathcal{E}$ is also adequate but with respect to a different behavioral refinement that relates closure values to primitive closure values (not constructed closures).

Using adequacy, we can derive whole-program correctness for our compiler by showing each transformation correct with respect to the logical relation and then transitively composing behavioral refinement (which is transitive). But this approach does not support compositional correctness.

The logical relations are also *compatible with linking*. This means that linking related modules with related modules yields related modules and it is captured by the following lemma. It holds for both relations, but we state it only for $\mathcal{E}$.

Lemma 5.6 (Linking compatibility). *Assume that* Compat Q_G Q_L. *The following holds.*

$$\begin{array}{l}
(\forall k\, \sigma_1\, \sigma_2,\ \mathcal{E}^k((e_1^{\mathrm{lib}}, \sigma_1), (e_2^{\mathrm{lib}}, \sigma_2))\ \{Q_L; Q_G\}) \quad \Rightarrow \\
(\forall k\, \sigma_1\, \sigma_2,\ \{x\} \vdash \mathcal{C}^k(\sigma_1, \sigma_2)\ \{Q_G\}\ \Rightarrow\ \mathcal{E}^k((e_1, \sigma_1), (e_2, \sigma_2))\ \{Q_L; Q_G\}) \quad \Rightarrow \\
\forall k\, \sigma_1\, \sigma_2,\ \mathcal{E}^k((\sigma_1, [x \mapsto e_1^{\mathrm{lib}}]e_1), (\sigma_2, [x \mapsto e_2^{\mathrm{lib}}]e_2))\ \{Q_L; Q_G\}
\end{array}$$

In the literature, this property is also referred to as *horizontal compositionality* [Neis et al. 2015; Song et al. 2019]. By combining linking compatibility with adequacy, we can derive a corollary: If a transformation inhabits the logical relation then it also satisfies the behavioral refinement for separate compilation we defined in the previous section.

Unfortunately, logical relations are not (in the general case) transitively composable. Therefore, there is no obvious way to specify the whole pipeline with a relation that is both adequate and compatible with linking.

5.1 Compositionality

To show the desired behavioral refinement for separate compilation we need to prove that the entire pipeline inhabits a relation that is adequate and compatible with linking. Since logical relations are not transitively composable, we cannot prove that the end-to-end pipeline inhabits a logical relation. However, we make a crucial observation: adequacy and compatibility with linking are closed under relation composition. We can devise a new relation that is adequate and compatible with linking by composing individual logical relations. Since each transformation is proved correct with respect to a logical relation, the entire pipeline can be easily shown to inhabit the composition of these logical relations. Therefore, we can show that it satisfies the correctness specifications for both whole-program and separate compilation.

We begin by defining the relation $\overline{\mathcal{E}}^+$ (fig. 6), the transitive closure of the reflexive logical relation (strengthened with some additional requirements). At the base case (rule Step), we require that two expressions are related with the top-level logical relation $\overline{\mathcal{E}}$. The local and global invariants are existentially quantified and do not appear in the top-level definition. We only require that they satisfy the compatibility rules (required for proving linking compatibility) and that the local invariant implies the desired upper bound (required for adequacy). Lastly, we require that the names of the free variables are preserved, which is required to show linking compatibility.

Lemma 5.7 (Linking compatibility of $\overline{\mathcal{E}}^+$). *Assume that* $\mathrm{closed}(e_1^{\mathrm{lib}})$ *and that* $\mathrm{fv}(e_1) \subseteq \{x\}$. *Then:*

$$\overline{\mathcal{E}}^+\ (e_1^{\mathrm{lib}}, e_2^{\mathrm{lib}}) \quad \Rightarrow \quad \overline{\mathcal{E}}^+\ (e_1, e_2) \quad \Rightarrow \quad \overline{\mathcal{E}}^+\ (([x \mapsto e_1^{\mathrm{lib}}]e_1), ([x \mapsto e_2^{\mathrm{lib}}]e_2))$$

$$\frac{\text{Compat } Q_G\ Q_L \qquad \text{UpperBound } Q_L \qquad \text{fv}(e_2) \subseteq \text{fv}(e_1) \qquad \overline{\mathcal{E}}\ (e_1, e_2)\ \{Q_L; Q_G\}}{\overline{\mathcal{E}}^+\ (e_1, e_2)}\ \textsc{Step}$$

$$\frac{\overline{\mathcal{E}}^+\ (e_1, e) \qquad \overline{\mathcal{E}}^+\ (e, e_2)}{\overline{\mathcal{E}}^+\ (e_1, e_2)}\ \textsc{Trans}$$

Fig. 6. The $\overline{\mathcal{E}}^+$ relation.

Lemma 5.7 implies that any program that has been transformed with a series of transformations that satisfy the $\mathcal{E}$ relation can be linked with a program that has been transformed with a (possibly different) series of transformations that satisfy $\mathcal{E}$. All transformations of the λ_{ANF} other than closure conversion satisfy the $\mathcal{E}$ relation. Because $\mathcal{E}$ is reflexive, the identity transformation also satisfies the $\mathcal{E}$ relation. For example, a program that has been inlined and shrink reduced can be safely linked with a program that had its dead parameters remove.

The proof of this property makes use of reflexivity. In the two premises, the number of transitivity steps of $\overline{\mathcal{E}}^+$ need not be the same. If the first premise has m transitivity steps and the second n transitivity steps the result will have $m + n$ transitivity steps. We explain this with an example.

Assume that we translate the client program e_1 to a program e_2 with two transformations and the library program e_3^{lib} to e_4^{lib} with only one transformation. We want to link e_2 with e_4^{lib}. For the client, we have that $\overline{\mathcal{E}}\ (e_1, e_1')\ \{Q_L^1; Q_G^1\}$ and $\overline{\mathcal{E}}\ (e_1', e_2)\ \{Q_L^2; Q_G^2\}$ for an intermediate program e_1' and some local and global invariants. For the library, we have that $\overline{\mathcal{E}}\ (e_3^{\text{lib}}, e_4^{\text{lib}})\ \{Q_L^3; Q_G^3\}$. We construct $\overline{\mathcal{E}}^+\ (([x \mapsto e_3^{\text{lib}}]e_1), ([x \mapsto e_4^{\text{lib}}]e_2))$ with the following transitivity steps:

(1) $\overline{\mathcal{E}}\ (([x \mapsto e_3^{\text{lib}}]e_1), ([x \mapsto e_3^{\text{lib}}]e_1'))\ \{Q_L^1; Q_G^1\}$
(2) $\overline{\mathcal{E}}\ (([x \mapsto e_3^{\text{lib}}]e_1'), ([x \mapsto e_3^{\text{lib}}]e_2))\ \{Q_L^2; Q_G^2\}$
(3) $\overline{\mathcal{E}}\ (([x \mapsto e_3^{\text{lib}}]e_2), ([x \mapsto e_4^{\text{lib}}]e_2))\ \{Q_L^3; Q_G^3\}$

Each one of these relations is proved by lemma 5.6 where one of the premises is obtained by reflexivity.

SepCompCert [Kang et al. 2016] uses the fact that compilation commutes with linking to derive a separate compilation theorem for CompCert. This technique also relies on reflexivity. By modifying syntactic simulations of optional transformations to be reflexive relations (so that they are inhabited by the identity transformation too) it allows separately compiled programs to use different optional optimizations. That is, a program may use an optional optimization or the identity transformation instead. Unlike ours, this technique is not compositional. The linking theorem is not stated with respect to a general relation but it depends on the particular pipeline. Hence, it has to be reproved when new transformations are added or when transformations are reordered.

Next, we define the $\overline{\mathcal{E}}^+_{\mathsf{CC}}$ relation (fig. 7), which is the top-level closure conversion relation $\overline{\mathcal{E}}_{\mathsf{CC}}$ composed with $\overline{\mathcal{E}}^+$ on the left and on the right. Our pipeline inhabits this relation.

As before, the local and global invariants are existentially quantified and we require that they satisfy Compat and UpperBound. The terms related with the closure conversion relation are also required to preserve free variables. Crucially, the $\overline{\mathcal{E}}^+_{\mathsf{CC}}$ relation is adequate and compatible with linking.

Lemma 5.8 (Adequacy of $\overline{\mathcal{E}}^+_{\mathsf{CC}}$). *If* $\overline{\mathcal{E}}^+_{\mathsf{CC}}\ (e_1, e_2)$ *and* $\text{closed}(e_1)$ *we have that* $e_1 \supseteq_{\mathsf{B}} e_2$.

$$\frac{\begin{array}{c}\text{Compat } Q_G\ Q_L \qquad \text{UpperBound } Q_L \\ \text{fv}(e_1') \subseteq \text{fv}(e_2') \\ \overline{\mathcal{E}}^{+}\ (e_1, e_1') \qquad \overline{\mathcal{E}}_{\text{CC}}\ (e_1', e_2')\ \{Q_L; Q_G\} \qquad \overline{\mathcal{E}}^{+}\ (e_2', e_2)\end{array}}{\overline{\mathcal{E}}^{+}_{\text{CC}}\ (e_1, e_2)}\ \text{Compose}$$

Fig. 7. The $\overline{\mathcal{E}}^{+}_{\text{CC}}$ relation.

Lemma 5.9 (Linking compatibility of $\overline{\mathcal{E}}^{+}_{\text{CC}}$). *Assume that* closed$(e_1^{\text{lib}})$ *and that* $\text{fv}(e_1) \subseteq \{x\}$. *Then:*

$$\overline{\mathcal{E}}^{+}_{\text{CC}}\ (e_1^{\text{lib}}, e_2^{\text{lib}}) \quad\Rightarrow\quad \overline{\mathcal{E}}^{+}_{\text{CC}}\ (e_1, e_2) \quad\Rightarrow\quad \overline{\mathcal{E}}^{+}_{\text{CC}}\ (([x \mapsto e_1^{\text{lib}}]e_1), ([x \mapsto e_2^{\text{lib}}]e_2))$$

We can easily derive a separate compilation theorem for any pipeline that is in the $\overline{\mathcal{E}}^{+}_{\text{CC}}$ relation; pipelines that are in the $\overline{\mathcal{E}}^{+}_{\text{CC}}$ relation need not use the same transformations. All transformations that satisfy $\mathcal{E}$ are optional; we can safely link modules regardless of whether these transformations are enabled or not during compilation. The only restriction is that pipeline encompass a closure conversion transformation that satisfies $\mathcal{E}_{\text{CC}}$—which is quite general. It would permit, for example, sophisticated closure representations such as Shao and Appel's safe-for-space hybrid flat/linked closures [Shao and Appel 1994]. This is another crucial difference with SepCompCert's theorem: SepCompCert only supports separate compilation with different *optional* transformations whereas our stronger compositional theorem supports separate compilation of programs with entirely different transformations as long as they satisfy the same logical relations.

Extension to cross-language setting. Even though we stay in the same λ_{ANF} language, our closure-conversion logical relation behaves as a cross-language logical relation. This technique supports separately compiling programs through pipelines that go through $n \geq 0$ representation changes that are surrounded by $n + 1$ chunks of same-representation passes, with each chunk comprising of an arbitrary number of passes. A top-level compositional relation is obtained by composing the adjacent cross-language relations, adding an intermediate reflexive relation for the IRs where same-language transformations happen. In our application $n = 1$ (closure conversion), but the principle generalizes to any n.

The rest of CertiCoq's pipeline is proved correct (or is in the process of being proved correct) with syntactic simulations. We can still derive an end-to-end theorem for separate compilation (with arbitrary λ_{ANF} optimizations), but it will not be compositional. The passes that are proved correct with a simulation will have to be the same in separately compiled programs.

6 COMPOSITIONAL CORRECTNESS

In this section, we establish the end-to-end correctness properties of our pipeline. First, we discuss the individual correctness proof of each transformation. Then, we show how to compose these correctness proofs to derive an end-to-end compositional compiler theorem for the whole pipeline.

6.1 Correctness of λ_{ANF} Transformations

We first discuss the resource invariants that we use in the correctness proof of each transformation. For now, we ignore fresh variable generation in order to simplify the type of transformations and the correctness statements. We make this explicit late in this section. Transformations that do not decrease the program steps satisfy a very simple invariant that states that the target steps are always greater or equal to the source steps. We define $\leq$ to be the invariant that maps two pairs of fuel and trace (c_1, t_1) and (c_2, t_2) to $c_1 \leq c_2$.

Closure conversion. The conclusion of the top-level theorem of the closure conversion transformation (denoted cc) states: $\overline{\mathcal{E}}_{\mathsf{CC}}\ (e, \mathrm{cc(e)})\ \{\leq; \leq\}$.

Lambda lifting, uncurrying, hoisting, DPE. These transformations satisfy the invariant $\leq$, therefore they satisfy $\overline{\mathcal{E}}\ (e, \mathsf{trans}(\mathrm{e}))\ \{\leq; \leq\}$, where trans is any of these transformations.

Inlining. The specification of inlining is trickier as this transformation can decrease the program steps. This is where we make use of the trace kept by our semantics. Recall that the trace is keeping track of how many function applications are executed by a program. First, consider that for each function that is inlined the target program will perform one or two steps less than the source. One step is removed because the target does not perform then function call. A second step might also be removed when we inline a let-bound call of a function that ends with return because the return statement will be removed in the inlined code. For example, consider the program $\mathtt{let}\ z = f\ x\ y\ \mathtt{in}\ e$ where f is defined (earlier in the program) to be $\mathtt{fun}\ f\ x\ y = \mathtt{let}\ w = \mathtt{Constr}(x, y)\ \mathtt{in}\ \mathtt{ret}(w)$. After inlining f the program becomes $\mathtt{let}\ z = \mathtt{Constr}(x, y)\ \mathtt{in}\ e$. The program before inlining consumes $3 + c$ units of fuel, where c is the fuel consumption of e, whereas the program after in lining consumes $1 + c$ units of fuel.

Let c_1 and t_1 be the fuel and trace (*i.e.*, the total number of function applications) of the source and c_2 and t_2 the fuel and trace of the target. Let G be the maximum number of calls that are inlined inside the body of a source function and L the total number of calls that are inlined in the top-level expression of the source program (without considering inlined calls in nested function definitions). The total number of inlined calls when the program runs is therefore upper bounded by $G * t_1 + L$, where t_1 is the total number of function calls executed by the source program, captured by the trace value. For each one of these inlined calls, the target might perform at most two evaluation steps less than the source program. So we have the bound $c_1 \leq c_2 + 2 * G * t_1 + 2 * L$. Using similar reasoning, we also have that $t_1 \leq t_2 + G * t_2 + L$ for the total number of calls performed by the source program. Because the total number of calls that a program performs is less or equal to the total number of steps, we have that $t_2 \leq c_2$. Using these three inequalities, we can derive a bound $c_1 \leq A * c_2 + B$ for some A and B that depend on G and L. This upper bound satisfies the requirements for divergence preservation of lemma lemma 4.2. We define:

$$\begin{aligned}
\mathsf{Q}_{\mathsf{inline}}\ L\ G\ (c_1, t_1)\ (c_2, t_2) &\overset{\mathrm{def}}{=} c_1 \leq c_2 + 2 * G * t_1 + 2 * L\ \wedge\ t_1 \leq t_2 + G * t_1 + L\ \wedge \\
&\phantom{\overset{\mathrm{def}}{=}}\ t_2 \leq c_2 \\
\mathsf{Q}^{\mathsf{top}}_{\mathsf{inline}}\ L\ G\ (c_1, t_1)\ (c_2, t_2) &\overset{\mathrm{def}}{=} c_1 \leq A(L, G) * c_2 + B(L, G)
\end{aligned}$$

For inlinining (inline) we show: $\overline{\mathcal{E}}\ (e, \mathsf{inline}(e))\ \{\mathsf{Q}_{\mathsf{inline}}\ \mathrm{G}\ \mathrm{G}; \mathsf{Q}_{\mathsf{inline}}\ \mathrm{G}\ \mathrm{G}\}$ where G is the maximum number of inlined calls in individual function bodies and the top-level expression (the transformation keeps track of that). Because the logical relation is monotonic in the local (but not the global) invariant we derive $\overline{\mathcal{E}}\ (e, \mathsf{inline}(e))\ \{\mathsf{Q}^{\mathsf{top}}_{\mathsf{inline}}\ \mathrm{G}\ \mathrm{G}; \mathsf{Q}_{\mathsf{inline}}\ \mathrm{G}\ \mathrm{G}\}$.

Shrinking. The shrinking transformation performs inlining as well as other static reductions and satisfies the same bound as inlining. We express shrinking (shrink) as the transitive closure of a transformation that performs one shrinking step (shrink_one). For the one-step shrinking, we prove $\overline{\mathcal{E}}\ (e, \mathsf{shrink\_one}(e))\ \{\mathsf{Q}_{\mathsf{inline}}\ 1\ 1; \mathsf{Q}_{\mathsf{inline}}\ 1\ 1\}$. By local invariant monotonicity, we derive $\overline{\mathcal{E}}\ (e, \mathsf{shrink\_one}(e))\ \{\mathsf{Q}^{\mathsf{top}}_{\mathsf{inline}}\ 1\ 1; \mathsf{Q}_{\mathsf{inline}}\ 1\ 1\}$. Observe that $G = L = 1$ since only one reduction is performed. Then, for the shrinking transformation we obtain $\overline{\mathcal{E}}^{+}\ (e, \mathsf{shrink}(\mathrm{e}))$.

6.2 Composition and End-to-End Correctness

So far, we presented λ_{ANF} transformations as total programs, but in reality they are partial programs: they may fail to produce an output for some inputs. Their type is

$$\mathsf{anf\_trans} \overset{\mathrm{def}}{=} \mathsf{exp} \to \mathsf{comp\_data} \to \mathsf{error\ exp} * \mathsf{comp\_data}$$

where exp is the type of λ_{ANF} terms and comp_data the type of the compilation state. The sum type `error` has two variants: programs can successfully return a term (written Ret e), or can return an error message (written Err s). Our transformations receive a compilation state comp_data as an argument threaded through the computation. They use it to generate fresh names for binders.

We show that whenever some input term e is well-scoped (written well_scoped(e)), meaning that e has unique bindings and its free variables are disjoint from its bound variables, then the transformation successfully produces an output term that is also well-scoped and semantically related to the input term with either $\overline{\mathcal{E}}^{+}$ or $\overline{\mathcal{E}}^{+}_{\mathsf{CC}}$. We assume that the next available free variable is strictly greater than the identifiers (free or bound) that are used in the input term (variables are represented as binary natural numbers). This implies that the freshly generated free variables are disjoint from the identifiers of the program, which is required for correctness and for preserving well-scopedness. The full top-level correctness specification of λ_{ANF} transformations is shown below.

Definition 6.1 (Correctness w.r.t. $\overline{\mathcal{E}}^{+}$). We say that a λ_{ANF} transformation *trans* $\in$ anf_trans is correct w.r.t to $\overline{\mathcal{E}}^{+}$ (written correct *trans*) if the following holds.

$$\begin{aligned}&\forall\, e\ c,\ \mathsf{well\_scoped}(e)\ \wedge\ \mathsf{max\_var}\ e < \mathsf{next\_var}\ c\ \to\\ &\quad \exists\, e' c',\ \mathit{trans}\ e\ c = (\mathsf{Ret}\ e', c')\ \wedge\ \mathsf{well\_scoped}(e')\ \wedge\ \mathsf{max\_var}\ e' < \mathsf{next\_var}\ c'\ \wedge\ \overline{\mathcal{E}}^{+}\ (e, e')\end{aligned}$$

Similarly, we define correct_cc, a predicate over λ_{ANF} transformations that has the same shape as correct, with the difference that it uses the $\overline{\mathcal{E}}^{+}_{\mathsf{CC}}$ relation.

λ_{ANF} transformations can be sequentially composed to derive complex λ_{ANF} transformations. Their correctness specifications can be composed as well. We have that:

- correct t_1 $\Rightarrow$ correct t_2 $\Rightarrow$ correct $(t_1;;t_2)$
- correct t_1 $\Rightarrow$ correct_cc t_2 $\Rightarrow$ correct_cc $(t_1;;t_2)$
- correct_cc t_1 $\Rightarrow$ correct t_2 $\Rightarrow$ correct_cc $(t_1;;t_2)$

We use these facts to derive end-to-end correctness.

The λ_{ANF} pipeline, comp_anf, is a λ_{ANF} transformation that takes an additional argument that determines which optional optimizations are enabled. We prove that whatever the choice of optional transformations is, the λ_{ANF} pipeline is correct w.r.t. to $\overline{\mathcal{E}}^{+}_{\mathsf{CC}}$.

Theorem 6.2 (Correctness of the λ_{ANF} pipeline.). $\forall\, \mathit{opt},$ correct_cc (comp_anf *opt*).

Using this top-level theorem for the λ_{ANF} pipeline, we can prove correctness of whole-program compilation and correctness for separate compilation as easy corollaries.

Corollary 6.3 (Correctness of whole-program compilation.). *Assume that* well_scoped(e_{src}) *and* closed(e_{src})*. Then, there exists* e_{trg} *such that* comp_anf *opt* e_{src} = Ret e_{trg} *and* $e_{\mathrm{src}} \supseteq_{\mathsf{B}} e_{\mathrm{trg}}$.

Corollary 6.4 (Correctness of separate compilation.). *Assume that* well_scoped($e^{\mathrm{lib}}_{\mathrm{src}}$), closed($e^{\mathrm{lib}}_{\mathrm{src}}$), well_scoped($e_{\mathrm{src}}$) *and* fv($e_{\mathrm{src}}$) $\subseteq \{x\}$*. Then, for all compilation options* opt_1 *and* opt_2*, there exist target programs* $e^{\mathrm{lib}}_{\mathrm{trg}}$ *and* e_{trg} *such that* comp_anf opt_1 $e^{\mathrm{lib}}_{\mathrm{src}}$ = Ret $e^{\mathrm{lib}}_{\mathrm{trg}}$ *and* comp_anf opt_2 e_{src} = Ret e_{trg} *and* $[x \mapsto e^{\mathrm{lib}}_{\mathrm{src}}]e_{\mathrm{src}} \supseteq_{\mathsf{B}} [x \mapsto e^{\mathrm{lib}}_{\mathrm{trg}}]e_{\mathrm{trg}}$

	No λ lifting	(New) Selective λ lifting	(Traditional) Nonselective λ lifting
sha	**1**±.012	**0.994**±.016	**0.994**±.019
color	**1**±.022	**0.984**±.019	**0.997**±.019
binom	**1**±.014	**0.999**±.020	**1.000**±.026
vs-hard	**1**±.008	**0.935**±.006	**0.951**±.007
vs-easy	**1**±.012	**0.928**±.008	**1.047**±.012

Fig. 8. Execution time of programs compiled with different lambda-lifting strategies, normalized to "no lambda lifting". Standard deviations shown are for 100 runs.

	CertiCoq (`clang`)	CertiCoq (CompCert)	OCaml Native Code
sha	**1**±.006	**1.343**±.007	**.730**±.010
color	**1**±.027	**1.202**±.029	∞
binom	**1**±.022	**4.462**±.040	**.235**±.007
vs-hard	**1**±.007	**1.522**±.029	**.457**±.006
vs-easy	**1**±.027	**1.456**±.007	**.328**±.005

Fig. 9. Execution time of OCaml native code generator versus λ_{ANF} with selective lambda lifting. Coq's built-in extraction produces illegal OCaml code for `color`, so OCaml could not compile it.

7 EVALUATION

Does selective lambda lifting (which pays attention to variable liveness) achieve better performance improvements than traditional non-selective lambda lifting? Does CertiCoq with the optimizing λ_{ANF} pipeline succeed in generating native-speed code that is comparable to standard extraction with optimizing OCaml native-code compilation?

Since there is no standard benchmark suite for Gallina, we devised four benchmarks to estimate answers to these questions:

sha: Secure Hash Algorithm 2 (SHA-256) computing the cryptographic hash of a 484-character string, as in Appel [2015].

color: A formally verified implementation of the Kempe/Chaitin graph-coloring algorithm [Chaitin et al. 1981] from Appel [2020], coloring a graph with 156 nodes and 1168 edges.

binom: A verified binomial queue implementation [Vuillemin 1978] from Appel [2020], constructing two priority queues with 1000 elements each, merging them, and finding the max.

vs-easy: VeriStar [Stewart et al. 2012], a formally verified decision procedure for a decidable fragment of separation logic, based on a state-of-the-art paramodulation algorithm [Navarro Pérez and Rybalchenko 2011], on an easy entailment.

vs-hard: VeriStar deciding validity of a harder entailment.

We then compared three versions of lambda-lifting (never, selective, always). For overall performance, we compared CertiCoq+LLVM (clang 10.0.1 with omit-frame-pointer), CertiCoq+CompCert (3.8), and Coq-extraction+ocamlopt (4.07.1). The experiments ran on a 2.5 GHz Intel Core i7.

Figure 8 demonstrates that (on these benchmarks) selective lambda lifting speeds up some programs and slows down none, which is not true of traditional nonselective lambda lifting.

CertiCoq's performance is not yet competitive with OCaml native code, as shown in Figure 9. We suspect that this is mainly because CertiCoq does not have a native code generator specialized for functional languages—it goes through C's calling convention[5] for all control flow, including loops ($\lambda^{\mathsf{C}}_{\mathsf{ANF}}$ would be suitable for direct generation of efficient machine language with the addition of a register allocator and, of course, a native code generator for each target machine.) Adding more optimizations in λ_{ANF} would probably further improve performance.

[5] We use clang with standard C calling conventions. Savary Bélanger et al. [2019] report that performance improves significantly when using clang with LLVM's "GHC calling convention" which has no callee-save registers and passes more arguments in registers than the standard. CompCert does not support nonstandard calling conventions.

8 RELATED WORK

We compare our work with proved-correct compilers and frameworks for compositional compiler correctness.

CakeML [Kumar et al. 2014; Tan et al. 2016] is an end-to-end verified compiler for a substantial subset of the ML language. Most closely related to our work is the ClosLang pipeline of CakeML [Owens et al. 2017; Tan et al. 2019]. Similar to the λ_{ANF} pipeline, ClosLang introduces calls to multi-argument functions and optimizes calls to known functions. CakeML uncurries both statically known and escaping functions. This is possible because the ClosLang semantics allows partial application (via mismatch semantics). Like λ_{ANF}, CakeML eliminates closures of statically known functions, but unlike λ_{ANF}, CakeML does not implement specialized closure allocation strategies for known functions with free variables. The ClosLang pipeline also performs two passes to track the flow of known function calls and annotate statically known functions and function calls. In λ_{ANF}, we do not need any additional flow analysis for our closure strategies. The ClosLang intermediate representation needs to distinguishe between ML-style and C-style calls, which is not needed in our setting.

ClosLang's verification framework uses untyped step-indexed logical relations. But unlike our framework, it does not address compositional compiler correctness. CakeML's theorem only applies to whole-program compilation. Divergence preservation is handled by requiring that two related programs have identical fuel consumption, which requires adding a `Tick` instruction to all intermediate representations of CakeML. With our work we show how this can be avoided.

Pilsner [Neis et al. 2015] is a multi-pass CPS compiler for an ML subset. Its compositional compiler correctness theorem is stronger than λ_{ANF}'s in that it applies to modules compiled from the same source, regardless of how are they compiled. Our technique works only on pipelines with the same series of intermediate languages. Pilsner is verified using *parametric intra-language simulations (PILS)*, a novel relational model that is adequate, compatible with linking, and transitive. But the metatheory of PILS, including its transitivity proof, is quite complex and required a lot of proof effort [Neis et al. 2015, Section 4]. We couldn't have used PILS to verify the λ_{ANF} pipeline because it does not support the eta-conversion rule used by our transformations (*e.g.*, uncurrying). A solution has been suggested [Hur et al. 2014], but has not been incorporated into PILS. Pilsner performs minimal optimizations; it does not uncurry functions or eliminate closures of known functions.

Œuf [Mullen et al. 2018] is a prototype compiler from Gallina (Coq's specification language) to Clight. Oeuf supports an impoverished subset of Coq: no user defined inductive types (just predefined bool, list, *etc.*), no pattern matching, no recursive functions, no dependently typed programs. CertiCoq supports full Gallina (except coinductive types), representable in λ_{ANF}. Œuf does not optimize the code; the authors state that it is a non-goal.

SepCompCert [Kang et al. 2016] extends CompCert's [Leroy 2009b] proof to separate compilation. Its theorem supports separate compilation of programs with different optional optimizations. As we explained in section 5, it is verified using a lightweight technique that is not compositional; the theorem about separate compilation is not derived from a general relation. Therefore, it has limitations that are not present in our work. First, SepCompCert requires modification to the statements and proofs of transformations. Second, in SepCompCert the linking theorem dependes on the pipeline and must be reproved for each variation of the pipeline. We prove it as a corollary of our compositional relation. Lastly, the linking theorem of SepCompCert applies only to optional transformations that can be replaced with the identity transformation. Our framework can be used to link programs that are compiled with different nonoptional transformations, such as two different implementations of closure conversion that may use different environment representations.

Compositional Compiler Correctness is a challenging problem that has been addressed in different contexts. There are many extensions of CompCert to compositional correctness: CompCompCert [Stewart et al. 2015], CompCertX [Gu et al. 2015; Wang et al. 2019], and CompCertM [Song et al. 2019]. CompCompCert and CompCertM's approaches are based on interaction semantics, which assumes the same memory representation in the source and target languages; this is not true for functional languages as the memory representation of closures changes between the source and target. It is therefore unlikely that these techniques would apply to CertiCoq ([Song et al. 2019, Section 8.1]). CompCertM uses closed simulations and assumes that unknown functions in the target are verified against deep specifications; it is not clear if these generalize to higher-order languages. Benton and Hur [Benton and Hur 2009] and Hur and Dreyer [Hur and Dreyer 2011] set up step-indexed logical relations between high- and low-level languages to study compositional equivalence of programs in these languages. But it is not clear how these frameworks could be used for multi-pass compilers, because the logical relations are not transitive. Perconti and Ahmed [Perconti and Ahmed 2014] use multi-language semantics [Matthews and Findler 2007] to support linking of source programs with arbitrary target code, which is outside the scope of this paper.

Logical Relations do not typically handle divergence preservation. Paraskevopoulou and Appel [Paraskevopoulou and Appel 2019] use a logical relation to prove their closure conversion transformation is correct and and safe for space. They parameterize their logical relation with relational pre- and postconditions that are similar our relational invariants and they use them to show both space safety and divergence preservation. However, they do not prove divergence preservation for transformations that reduce the amount of fuel, which is a significantly more challenging problem. Their framework does not address compositional compiler correctness.

9 CONCLUSION

We presented a novel verification framework that supports lightweight compositional compiler correctness. In particular, we show how logical relations, which generally do not support vertical compositionality and cannot be used to show compositional compiler correctness for multi-pass pipelines, can be composed to derive a compositional compiler correctness theorem for multi-pass pipelines. The only restriction that we impose is that the pipelines should go through the same changes in representation. Furthermore, we show how this framework can be used to show divergence preservation for transformations, overcoming some complications of previous approaches.

We applied this framework to prove correct an optimizing multi-pass pipeline on an A-normal form intermediate representation for the CertiCoq compiler. Our optimizing pipeline compiles purely functional programs to a flat-scope programs without closures that can be easily code-generated. We expect that future compiler verification efforts will benefit from the ideas of our verification framework and the design of our pipeline.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation under Grant No. CCF-1521602 and Grant No. 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. The first author was supported by Computing Innovation Fellows 2020 Project during part of this research. We wish to thank the anonymous reviewers for their insightful comments and Norman Ramsey for his valuable feedback in the final revision of this paper.

REFERENCES

Umut A. Acar, Amal Ahmed, and Matthias Blume. 2008. Imperative Self-Adjusting Computation. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 309–322. https://doi.org/10.1145/1328438.1328476

Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. 1986. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction* (Palo Alto, California, USA) *(SIGPLAN '86)*. Association for Computing Machinery, New York, NY, USA, 219–233. https://doi.org/10.1145/12276.13333

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Programming Languages and Systems*, Peter Sestoft (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 69–83.

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

Andrew W. Appel and Trevor Jim. 1997a. Shrinking lambda expressions in linear time. *Journal of Functional Programming* 7, 5 (Sept. 1997), 515–540. https://doi.org/10.1017/S0956796897002839

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

Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2, Article 7 (April 2015), 31 pages. https://doi.org/10.1145/2701415

Andrew W. Appel. 2020. Verified Functional Algorithms. Version 1.4, http://softwarefoundations.cis.upenn.edu.

Andrew W. Appel and Trevor Jim. 1997b. Shrinking Lambda Expressions in Linear Time. *J. Funct. Program.* 7, 5 (Sept. 1997), 515–540.

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (Sept. 2001), 657–683. https://doi.org/10.1145/504709.504712

Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2019. Formal Verification of a Constant-Time Preserving C Compiler. *Proc. ACM Program. Lang.* 4, POPL, Article 7 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371075

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, Step-Indexing and Compiler Correctness. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/1596550.1596567

Nick Benton, Andrew Kennedy, Sam Lindley, and Claudio Russo. 2005. Shrinking Reductions in SML.NET. In *Implementation and Application of Functional Languages*, Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 142–159.

Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. 1981. Register allocation via coloring. *Computer languages* 6, 1 (1981), 47–57.

Adam Chlipala. 2010. A Verified Compiler for an Impure Functional Language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 93–106. https://doi.org/10.1145/1706299.1706312

Zaynah Dargaye and Xavier Leroy. 2007. Mechanized Verification of CPS Transformations. In *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence and Reasoning* (Yerevan, Armenia) *(LPAR'7)*. Springer-Verlag, Berlin, Heidelberg, 211–225.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

Pascal Fradet and Daniel Le Métayer. 1991. Compilation of Functional Languages by Program Transformation. *ACM Trans. Program. Lang. Syst.* 13, 1 (Jan. 1991), 21–51. https://doi.org/10.1145/114005.102805

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and Provable Local Capability Revocation Using Uninitialized Capabilities. *Proc. ACM Program. Lang.* 5, POPL, Article 6 (Jan. 2021), 30 pages. https://doi.org/10.1145/3434287

Sebastian Graf and Simon Peyton Jones. 2019. Selective Lambda Lifting. *CoRR* abs/1910.11717 (2019). arXiv:1910.11717 http://arxiv.org/abs/1910.11717

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

R. J. M. Hughes. 1982. Super-Combinators: A New Implementation Method for Applicative Languages. In *In Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*. ACM, 1–10.

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke Logical Relation Between ML and Assembly. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*.

ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1926385.1926402

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

Neis Georg Hur, Chung-Kil, Derek Dreyer, and Viktor Vafeiadis. 2014. *Parametric Bisimulations: A Logical Step Forward.* Technical Report.

Thomas Johnsson. 1985. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture* (Nancy, France). Springer-Verlag, Berlin, Heidelberg, 190–203.

Simon L. Peyton Jones. 1996. Compiling Haskell by program transformation: A report from the trenches. In *Programming Languages and Systems — ESOP '96*, Hanne Riis Nielson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–44.

Simon Peyton Jones and André Santos. 1995. Compilation by Transformation in the Glasgow Haskell Compiler. In *Functional Programming, Glasgow 1994*, Kevin Hammond, David N. Turner, and Patrick M. Sansom (Eds.). Springer London, London, 184–204.

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. *SIGPLAN Not.* 51, 1 (Jan. 2016), 178–190. https://doi.org/10.1145/2914770.2837642

R. Kelsey and P. Hudak. 1989. Realistic Compilation by Program Transformation (Detailed Summary). In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 281–292. https://doi.org/10.1145/75277.75302

Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. ACM, New York, NY, USA, 177–190. https://doi.org/10.1145/1291151.1291179

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. *SIGPLAN Not.* 49, 1 (Jan. 2014), 179–191. https://doi.org/10.1145/2578855.2535841

Xavier Leroy. 2009a. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://xavierleroy.org/publi/compcert-CACM.pdf

Xavier Leroy. 2009b. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. http://xavierleroy.org/publi/compcert-backend.pdf

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2020. *The OCaml system release 4.11.* Available electronically at https://coq.inria.fr/refman.

Xavier Leroy and Hervé Grall. 2009. Coinductive Big-Step Operational Semantics. *Inf. Comput.* 207, 2 (Feb. 2009), 284–304. https://doi.org/10.1016/j.ic.2007.12.004

Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/1190216.1190220

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. 2017. Compiling Without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 482–494. https://doi.org/10.1145/3062341.3062380

Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: Minimizing the Coq Extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 172–185. https://doi.org/10.1145/3167089

Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2011. Separation logic + superposition calculus = heap theorem prover. In *PLDI*. 556–566.

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* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 166–178. https://doi.org/10.1145/2784731.2784764

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 589–615.

Scott Owens, Michael Norrish, Ramana Kumar, Magnus O. Myreen, and Yong Kiam Tan. 2017. Verifying Efficient Function Calls in CakeML. *Proc. ACM Program. Lang.* 1, ICFP, Article 18 (Aug. 2017), 27 pages. https://doi.org/10.1145/3110262

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

Zoe Paraskevopoulou and Anvay Grover. 2021. Compiling with Continuations, Correctly. (2021). Under submission.

Daniel Patterson and Amal Ahmed. 2019. The next 700 Compiler Correctness Theorems (Functional Pearl). *Proc. ACM Program. Lang.* 3, ICFP, Article 85 (July 2019), 29 pages. https://doi.org/10.1145/3341689

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-Language Semantics. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 128–148. https://doi.org/10.1007/978-3-642-54833-8_8

G.D. Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1, 2 (1975), 125 – 159. https://doi.org/10.1016/0304-3975(75)90017-1

Tahina Ramananandro, Zhong Shao, Shu-Chun Weng, Jérémie Koenig, and Yuchen Fu. 2015. A Compositional Semantics for Verified Separate Compilation and Linking. In *Proceedings of the 2015 Conference on Certified Programs and Proofs* (Mumbai, India) *(CPP '15)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2676724.2693167

Leonardo RodrÃŋguez, Miguel Pagano, and Daniel Fridlender. 2016. Proving Correctness of a Compiler Using Step-indexed Logical Relations. *Electronic Notes in Theoretical Computer Science* 323 (07 2016), 197–214. https://doi.org/10.1016/j.entcs.2016.06.013

Andre Santos. 1995. *Compilation by transformation for non-strict functional languages.* Ph.D. Dissertation. University of Glasgow. https://www.microsoft.com/en-us/research/publication/compilation-transformation-non-strict-functional-languages/

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* (Namur, Belgium) *(PPDP '17)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/3131851.3131859

Olivier Savary Bélanger, Matthew Z. Weaver, and Andrew W. Appel. 2019. Certified Code Generation from CPS to C. (Oct. 2019). https://www.cs.princeton.edu/~appel/papers/CPStoC.pdf.

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

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

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2019. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371091

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 8 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371076

Guy L. Steele. 1978. *Rabbit: A Compiler for Scheme.* Technical Report. USA.

Gordon Stewart, Lennart Beringer, and Andrew W. Appel. 2012. Verified Heap Theorem Prover by Paramodulation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (Copenhagen, Denmark) *(ICFP '12)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2364527.2364531

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 275–287. https://doi.org/10.1145/2676726.2676985

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML. In *International Conference on Functional Programming (ICFP)*. ACM Press, 60–73. https://doi.org/10.1145/2951913.2951924 Invited to special issue of Journal of Functional Programming.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29, Article e2 (2019). https://doi.org/10.1017/S0956796818000229

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2017. A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of runST. *Proc. ACM Program. Lang.* 2, POPL, Article 64 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158152

Jean Vuillemin. 1978. A Data Structure for Manipulating Priority Queues. *Commun. ACM* 21, 4 (April 1978), 309–315. https://doi.org/10.1145/359460.359478

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290375