Compiling with Continuations, Correctly

ZOE PARASKEVOPOULOU, Northeasten University, USA
ANVAY GROVER, University of Wisconsin-Madison, USA

In this paper we present a novel simulation relation for proving correctness of program transformations that combines syntactic simulations and logical relations. In particular, we establish a new kind of simulation diagram that uses a small-step or big-step semantics in the source language and an untyped, step-indexed logical relation in the target language. Our technique provides a practical solution for proving semantics preservation for transformations that do not preserve reductions in the source language. This is common when transformations generate new binder names, and hence \( \alpha \)-conversion must be explicitly accounted for, or when transformations introduce administrative redexes. Our technique does not require reductions in the source language to correspond directly to reductions in the target language. Instead, we enforce a weaker notion of semantic preorder, which suffices to show that semantics are preserved for both whole-program and separate compilation. Because our logical relation is transitive, we can transition between intermediate program states in a small-step fashion and hence the shape of the proof resembles that of a simple small-step simulation.

We use this technique to revisit the semantic correctness of a continuation-passing style (CPS) transformation and we demonstrate how it allows us to overcome well-known complications of this proof related to \( \alpha \)-conversion and administrative reductions. In addition, by using a logical relation that is indexed by invariants that relate the resource consumption of two programs, we are able show that the transformation preserves diverging behaviors and that our CPS transformation asymptotically preserves the running time of the source program. Our results are formalized in the Coq proof assistant. Our continuation-passing style transformation is part of the CertiCoq compiler for Gallina, the specification language of Coq.

CCS Concepts: • Software and its engineering → Formal software verification.

Additional Key Words and Phrases: continuation-passing style, compiler correctness, logical relations, simulations

ACM Reference Format:
Zoe Paraskevopoulou and Anvay Grover. 2021. Compiling with Continuations, Correctly. Proc. ACM Program. Lang., Vol. 5, No. OOPSLA, Article 114 (October 2021), 29 pages. https://doi.org/10.1145/3485491

1 INTRODUCTION

Simulation relations are a well-known proof technique for showing that program transformations preserve the semantics of a source program. The exact shape of the simulation relation can vary depending on the languages and the transformation in question. A simple forward simulation relation states that when a behavior is exhibited by the source program, it may also exhibited by the target program. For deterministic languages, a forward simulation implies the transformation preserves diverging behaviors and that our CPS transformation asymptotically preserves the running time of the source program. Our results are formalized in the Coq proof assistant. Our continuation-passing style transformation is part of the CertiCoq compiler for Gallina, the specification language of Coq.

Authors’ addresses: Zoe Paraskevopoulou, Northeastern University, USA, z.paraskevopoulou@northeastern.edu; Anvay Grover, University of Wisconsin-Madison, USA, anvayg@cs.wisc.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2021 Copyright held by the owner/authors.
2475-1421/2021/10-ART114
https://doi.org/10.1145/3485491
However, a simple simulation diagram like that is not possible to establish for every translation as reductions in the source may not always correspond to reductions in the target. A common reason for that is $\alpha$-conversion: if the program transformation introduces new binders then the simple simulation diagram does not hold anymore. The target program $\llbracket s_{\text{src}} \rrbracket$ reduces to a term $s_{\text{trg}}$ that is $\alpha$-equivalent—but not syntactically equal—to $\llbracket s'_{\text{src}} \rrbracket$. Therefore, it is necessary to reason explicitly up to $\alpha$-equivalence, which can be daunting. Another complication arises when a transformation introduces administrative redexes, i.e. redexes that can be reduced at compile time. In this case, the target program $\llbracket s_{\text{src}} \rrbracket$ reduces to a term $s_{\text{trg}}$ that is not equal to $\llbracket s'_{\text{src}} \rrbracket$, but instead $\llbracket s'_{\text{src}} \rrbracket$ may be reduced to $s_{\text{trg}}$.

In this paper we present a novel way of structuring simulation relations that avoids complications of traditional simulation relations. In particular, we state that when $s_{\text{src}}$ reduces to $s'_{\text{src}}$ then $\llbracket s_{\text{src}} \rrbracket$ and $\llbracket s'_{\text{src}} \rrbracket$ are logically related. We use an untyped, step-indexed logical relation that is transitive. This lets us use the logical relation to transition between intermediate program states like using a small-step semantics. Therefore, the shape of the simulation diagram as well as its proof is similar to that of a standard simulation. Because $\alpha$-equivalent terms are logically related, there is no need to explicitly account for $\alpha$-conversion. In addition, even when reductions in the source do not directly correspond to reductions in the target, the simulation relation still holds as both target terms ought to compute semantically the same results. To our knowledge, this is the first time a logical relation has been used in this way. We use this technique to show correctness for the CPS transformation that is used by the CertiCoq compiler [Anand et al. 2017].

Translation to continuation-passing style is one of the most thoroughly investigated program transformations. CPS transformations have important theoretical implications [Griffin 1989; Plotkin 1975] and practical uses in compiler construction [Adams et al. 1986; Appel 1992; Kennedy 2007; Steele 1978]. The first proof that CPS transformation preserves semantics is provided by Plotkin [1975] in his seminal paper about call-by-value and call-by-name CPS translations. With a simulation argument, he shows that call-by-value lambda calculus can be simulated with call-by-name lambda calculus and vice-versa, establishing that CPS translations encode an evaluation strategy into the syntax of the term. However, the administrative redexes that are introduced by the CPS transformation complicate the simulation diagram, as reductions in the source program do not correspond to reductions in the target. Plotkin introduces an auxiliary colon translation that performs administrative reductions and establishes a more complicated simulation diagram. Establishing a simulation relation using big-step semantics [Dargaye and Leroy 2007] avoids the difficulties with administrative reductions, but the correctness result applies only to terminating source programs and cannot be used to show divergence preservation. One-pass CPS translations, like those presented by Danvy and Filinski [1992] or Danvy and Nielsen [2003] do not introduce administrative reductions and hence do not suffer from this issue. But even one-pass transformations may fail to preserve reductions, in particular in presence of a let-construct. Pottier [2017] resorts to establishing a more complicated simulation argument that composes call-by-value reduction with parallel call-by-value reduction and Yamada and Asai [2018] resort to defining a translation-dependent equality relation.

Reasoning explicitly about binder names is another major source of complication in correctness proofs of CPS transformations, and authors that have attempted mechanizing such proofs advocate for different approaches to deal with binders. (Arguably, the mechanization of the correctness of CPS can serve as an additional benchmark for the POPLmark challenge [Aydemir et al. 2005].) Minamide and Okuma [2003] use named binders to mechanize several CPS transformations in backward simulation and it is therefore preferred. In this paper we focus on forward simulations since the languages we consider are deterministic. The technique we present can be used to establish a backward simulation too.
Isabelle/HOL and they take explicit account of $\alpha$-conversion in their simulation relations. Dargaye and Leroy [2007] mechanize the correctness proof Plotkin’s [1975] and Danvy and Nielsen’s [2003] CPS transformations in the Coq proof assistant using De Bruijn indices to represent binders to avoid reasoning explicitly about $\alpha$–equivalence. But in order to provide an efficient implementation of the CPS transformation in De Bruijn representation, they use a nonstandard syntax with two kinds of De Bruijn indices, one for source variables and one for continuation variables. Pottier [2017] also uses De Bruijn indices to mechanize in Coq a variant of Danvy and Filinski’s [1992] CPS transformation. He first proves correct an inefficient CPS transformation, with a nonstandard simulation diagram. He provides an efficient but, according to the author, hard to discover higher-order implementation of the CPS transformation that he proves equivalent to the inefficient implementation. In both cases, dealing with De Bruijn indices introduces implementation difficulties and requires a number of technical lemmas about lifting and shifting operations. Other authors [Chlipala 2008; Tian 2006; Yamada and Asai 2018] use (parametric) higher-order abstract syntax to avoid difficulties with $\alpha$-conversion.

In this paper, we avoid many of the technical complications of prior work on CPS correctness proofs by using our novel simulation relation in order to show that a CPS transformation preserves observable behaviors for both whole-program and separate compilation. We show that whatever the choice of binder names is, the CPS transformation produces terms are logically related. We therefore reason up to logical relatedness, without needing to explicitly account for $\alpha$-conversion. Our technique can handle transformations that introduce administrative reductions (or otherwise do not preserve reductions in the source language) without the need to resort to auxiliary translations or more complicated notions of reductions. The technique is not tied to a particular CPS transformation; it rather provides a systematic way to deal with common difficulties when establishing simulation diagrams while sticking to a straightforward named binder representation that is the most common in compiler construction. The technique itself can be used in the correctness proofs of other transformations and the machinery we employ is reusable in other contexts as well (notably, the logical relation that we use has been used to show correctness of seven other CertiCoq transformations [Paraskevopoulos et al. 2021]). By using a logical relation that is parameterized by invariants that relate the consumed resources of two programs, we are able to show that our CPS transformation preserves both divergence and the running time of the source program.

The CPS transformation that we consider is part of the CertiCoq verified compiler for Coq. Our source language is an untyped, call-by-value lambda calculus extended with let binding, constructors, pattern matching, and mutually recursive functions. Our target language supports the same language constructs but it is a CPS intermediate representation. Our CPS translation is a simpler version of Danvy and Nielsen’s transformation, that avoids some—but not all—administrative redexes. The rest of administrative redexes are removed by the shrink-reduction transformation of CertiCoq [Appel and Jim 1997; Savary Bélanger and Appel 2017]. We make this choice to keep the implementation and verification simpler, as Danvy and Nielsen’s one-pass transformation distinguishes 4 cases for function application and $2^n$ cases for constructors with arity $n$.

**Summary of contributions.**

- We present a novel proof technique that establishes a simulation between a source-level semantics of the source (small-step or big-step) and and a target-level logical relation. This simulation relation resembles small-step simulation diagrams in its simplicity, but unlike those, it holds for transformations that do not preserve reductions and does not require explicit reasoning about $\alpha$-equivalence.
- We prove that CPS transformation preserves asymptotically the running time of source programs. To our knowledge, this is the first proof of such a result for a CPS transformation.
Our correctness results are formalized in the Coq proof assistant. In the first part of the paper (section 2 and section 3), we describe our technique using simple lambda calculus. In the second part (section 4 through section 7), we describe the mechanization of our proof technique in CertiCoq. We conclude with a brief history of various correctness results for CPS transformations (section 8).

2 WARM-UP: LAMBDA CALCULUS

To begin with, let us consider ordinary lambda calculus. To illuminate the complexities that arise during traditional correctness proofs of CPS, we will define a simple CPS transformation and we will attempt to prove it correct using a small-step simulation. Then, we will demonstrate how our solution deals with these problems. The syntax we consider uses named binders and is shown below.

We represent variables as natural numbers, i.e., $\text{Var} \overset{\text{def}}{=} \mathbb{N}$. We distinguish between expressions and values, which are mutually defined.

\[
\begin{align*}
x \in \text{Var} & \quad e \in \text{Exp} ::= v \mid e_1 e_2 \quad v \in \text{Val} ::= \lambda x. e \mid x
\end{align*}
\]

We define a simple CPS transformation that takes explicit account of fresh variable generation. This CPS transformation is a smaller variation of Danvy and Nielsen [2003] that eliminates some, but not all, administrative redexes of Plotkin’s transformation. We define $[e]_k$ to be the CPS translation of expression $e$ with $k$ being the current continuation. We define $\langle v \rangle$ to be the CPS translation of value $v$. The programs are written in state-passing style (which we hide under a monad) to account for fresh variable generation. The state keeps track of the next available fresh variable. The type of $[e]_k$ is $\text{Var} \rightarrow (\text{Exp} \times \text{Var})$ and the type of $\langle v \rangle$ is $\text{Var} \rightarrow (\text{Val} \times \text{Var})$. At the top-level, the state is initialized with the successor of the largest variable used in the source program and it is increased by one every time we pick a fresh name. We define the meta-language function $\text{fresh} \overset{\text{def}}{=} \text{fun } v \Rightarrow (v, v + 1)$.

\[
\begin{align*}
\langle v \rangle_k & = v' \leftarrow \langle v \rangle; \text{ret } (k \ v') \\
[e_1 e_2]_k & = m \leftarrow \text{fresh}; n \leftarrow \text{fresh}; \quad \langle x \rangle = x \\
& e_2' \leftarrow [e_2]_{\lambda n. m \ k \ n}; [e_1]_{\lambda m. e_2'} \\
\langle \lambda x. e \rangle & = k \leftarrow \text{fresh}; \quad e' \leftarrow [e]_k; \text{ret } (\lambda k. \lambda x. e')
\end{align*}
\]

We wish to prove that this CPS translation preserves semantics, and we will try to do this using a standard small-step simulation argument. Let us, for a moment, ignore binder names and reason implicitly up to $\alpha$-equivalence. We will try to prove that whenever the source term takes a step then the CPS-translated term steps with one or more reductions to the CPS-translation of the reduced source term: $e \rightarrow e' \Rightarrow [e]_k \rightarrow^+ [e']_k$.

We write $\rightarrow$ for the standard call-by-value reduction in lambda calculus and $\rightarrow^+$ (resp. $\rightarrow^*$) for its transitive (resp. reflexive transitive) closure. The proof relies on a simple substitution lemma $[e]_k(\langle v \rangle/x) = [e[v/x]]_{k[\langle v \rangle/x]}$, which follows easily when we ignore binder names.

2.1 Failure of the Simple Simulation

To prove the statement, we proceed by induction on the source term. To see where the simple simulation fails, consider the case when $e \equiv v_1 e_2$ and $v_1 e_2 \rightarrow v_1 e_2'$ with $e_2 \rightarrow e_2'$. We need to show that $[v_1 e_2]_k \rightarrow^+ [v_1 e_2']_k$. We have that $[v_1 e_2]_k = [v_1]_{\lambda n. [e_2]_{\lambda n. m \ k \ n}} = (\lambda n. [e_2]_{\lambda n. m \ k \ n}) \langle v_1 \rangle$. This term has administrative redexes, i.e. useless redexes that are generated by the transformation and can be reduced at compile time. By performing the administrative reductions and using the substitution lemma we obtain the term $[e_2]_{\lambda n, \langle v_1 \rangle} k \ n$. This, in turn, reduces to $[e_2']_{\lambda n, \langle v_1 \rangle} k \ n$ by the induction hypothesis. However, we have that $[v_1 e_2']_k = (\lambda n. [e_2']_{\lambda n. m \ k \ n}) \langle v_1 \rangle$, which is not equal to the previous term, but it reduces to it after performing the administrative reductions.
The standard technique to overcome this issue, introduced by Plotkin [1975], is to define the so called colon translation that reduces some of the administrative redexes. The colon translation—which is itself a CPS transformation—is shown below. As in our previous definitions we explicitly account for fresh binder generation. The transformation removes some of the administrative redexes that are introduced when the subterms of an application are value forms.

\[
v : k = v' \leftarrow ([v]); \text{ret}(k \cdot v')
\]

\[
v_1 v_2 : k = v'_1 \leftarrow ([v_1]); v'_2 \leftarrow ([v_2]); \text{ret}(v'_1 \cdot k \cdot v'_2)
\]

\[
v_1 e_2 : k = n \leftarrow \text{fresh}; v'_1 \leftarrow ([v_1]); e_2 : \lambda n. v'_2 \cdot k \cdot n
\]

\[
e_1 e_2 : k = m \leftarrow \text{fresh}; e'_2 \leftarrow ([e_2])\lambda n. m \cdot k \cdot n; e_1 : \lambda m. e'_2
\]

Then to obtain a simulation diagram, we need to show that:

(a) \([e]_k \rightarrow^* e : k\), and

(b) if \(e \rightarrow e'\) then \([e]_k \rightarrow^* e' : k\).

Then, because the colon translation coincides with the expression translation on values, we have that if \(e \rightarrow^* v\), then \([e]_k \rightarrow^* [v]_k\).

The simulation proof that failed previously now succeeds for the color translation. We consider again the case when \(v_1 e_2 \rightarrow v_1 e'_2\) with \(e_2 \rightarrow e'_2\). We need to show that \([v_1 e_2]_k \rightarrow^* v_1 e'_2 : k\). As before, we have that \([v_1 e_2]_k\) reduces to \([e_2]\lambda n.[v_1] k \cdot n\). By the induction hypothesis, this reduces to \(e'_2 : \lambda n.[v_1] k \cdot n\), which is exactly \(v_1 e'_2 : k\).

**The \(\alpha\)-equivalence problem.** Let us now try to do the same proof while also taking explicit account of binder names. We define \([e]^x\lambda v /x = \text{fst}([e][/x])\), \((v)^x = \text{fst}(([v] x))\), and \(e : k = \text{fst}(e : k i)\) (note that \(\text{fst}\) and application are meta-language operations here). That is, \([e]_k^x\) is the CPS-translation of \(e\) using \(x\) as the initial state for fresh name generator. We say that a variable \(x\) is fresh with respect to an expression \(e\) (resp. value \(v\)) if it is greater than all free or bound variables of \(e\) (resp. \(v\))—recall that variables are represented as natural numbers. In what follows we assume that \(x\) is always fresh w.r.t. to \(e\) and the continuation term \(k\). We make this explicit later.

We could try to prove that \([e]_k^x \rightarrow^* e : k\) but that simply does not hold. \([e]_k^x\) will reduce to a term that is equal to \(e : y : k\) for some variable \(y\) that is not necessarily equal to \(x\).

Trying to generalize \(x\) in the theorem statement does not help much. Proving that there exist \(y\), such that \([e]_k^x \rightarrow^* e : y : k\) fails because the induction hypothesis is not strong enough. The arbitrary witness for \(y\) provided by the induction hypothesis for the subterms of \(e\) is not the one required to complete the proof.

The solution is to adjust the statements to explicitly account for \(\alpha\)-equivalence. Then, the above statements become:

(a) There exists \(e'\), such that \([e]_k^x \rightarrow^* e'\) and \(e : x : k \approx_\alpha e'\).

(b) If \(e \rightarrow e'\) then there exists \(e''\) such that \(e : x : k \rightarrow^* e''\) and \(e' : x : k \approx_\alpha e''\).

Reasoning explicitly about \(\alpha\)-equivalence is complicated. First observe that the substitution lemma does not hold anymore in its simple form; rather it becomes \([e]_k[(v)/x] \approx_\alpha [e[v/x]]_k[(v)/x]\).

We consider again the proof of statement (a) for the case when \(v_1 e_2 \rightarrow v_1 e'_2\) with \(e_2 \rightarrow e'_2\). We need to show that \([v_1 e_2]_k^x\) reduces to a term that is \(\alpha\)-equivalent to \(v_1 e_2 : x : k\). We have that

\[
[v_1 e_2]_k^x = [v_1]_y^x \lambda m.[e_2]_{x+2}^{x+2} \cdot \lambda n.[e_1]_{x+1}^{x+1} k \cdot n\]

where \(m = x, n = x + 1, y\), and \(y\) is the next available fresh name after the call to \([e_2]_{x+2} \lambda n.[e_1]_{x+1}^k\). After performing the administrative reductions, we obtain the term \([e_2]_{x+2} \lambda n.[v_1] y \cdot k \cdot n\) which, from the substitution lemma, is \(\alpha\)-equivalent to \([e_2]_{x+2} \lambda n.[v_1] y \cdot k \cdot n\). By the induction hypothesis, we obtain a term \(e''\) such that \([e_2]_{x+2} \lambda n.[v_1] y \cdot k \cdot n\) reduces to \(e''\) and \(e'' \approx_\alpha e_2 : x+2 \cdot \lambda n.[v_1] y \cdot k \cdot n\). But we have that \(v_1 e_2 : x : k = e_2 : y' : \lambda n.[v_1] y+1 \cdot k \cdot n\), where \(n = x\), and \(y'\) is the next available fresh variable after the call to \([v_1]_{y+1}\). To complete the proof...
we must show that $e_2 : x^{i+2} \lambda n.([v_1]^y) k n$ is $\alpha$-equivalent to $e_2 : y^i \lambda m.([v_1]^{x+1} k m$. Proving this requires setting up lemmas that allow us to reason about $\alpha$-equivalence. We must also show that $[e]_k^x \approx_\alpha [e']_k^y$ for all fresh $x$ and $y$ (and similarly for the value and colon translation).

Many mechanization efforts avoid reasoning explicitly up to $\alpha$-equivalence by using other binder representations. Dargaye and Leroy [2007] mechanize the correctness proofs of Plotkin’s CPS transformation and a one-pass variation of Danvy and Nielsen’s transformation. They use De Bruijn indices but this creates several other complications. First, it is not clear how to implement the CPS translation efficiently using De Bruijn indices because of the shifting and lifting operations that need to be applied to subterms. They manage to provide an efficient transformation but they resort to a De Bruijn representation that uses different name spaces for user and continuation variables. De Bruijn indices also require more complicated substitution lemmas (for example, that lifting of De Bruijn indices commutes with substitution). The authors observe that the one-pass transformation does not commute with substitution and therefore they are unable to provide a direct correctness proof. Instead, they show that Plotkin’s transformation reduces, using parallel call-by-value $\beta$-reduction, to the one-pass transformation.

Even if we were willing to commit to De Bruijn representation for the CPS pass of the compiler, De Bruijn indices would be very difficult to maintain throughout our compiler pipeline. Straightforward implementations of compiler passes using De Bruijn indices are inefficient because of the lifting operations; adjusting them to operate on De Bruijn indices in an efficient way would lead to complicated algorithms and data structures. Therefore, we would have to introduce named binders after the CPS transformation and deal with the complications of $\alpha$-equivalence during the correctness proof of that pass.

In the remainder of this section we show how using a logical relation can simplify both the simulation diagram of this proof and dealing with named binder representations without explicitly accounting for $\alpha$-equivalence.

### 2.2 Using Logical Relations

The above proof became complicated by the fact that there is no sequence of reductions between $[e]_k^x$ and $[e']_k^y$ when $e \rightarrow e'$. Our observation is that despite there being no sequence of reductions between the two terms, they still compute semantically related results. Indeed, we can define a simple, untyped, step-indexed logical relation that captures the fact that two terms are semantically related. We will use this relation to state and prove a simpler simulation diagram for our CPS transformation.

We define a step-indexed expression relation $\preceq_i$ that relates two expressions at step index $i$ if whenever the first expression reduces to some value $v_1$ with some number of steps $c_1 \leq i$, then the second expression also reduces to some value $v_2$ and the two values are related by the value relation for the remaining $i - c_1$ steps. The value relation $\preceq_i$ asserts that two lambda abstractions are related if they map any two arguments related at strictly smaller step indices to related results.

$$
\begin{align*}
\lambda x.e_1 \preceq_i \lambda y.e_2 & \iff \forall v_1, v_2, j (j < i), v_1 \preceq_j v_2 \Rightarrow e_1[v_1/x] \preceq_j e_2[v_2/y]
\end{align*}
$$

(Value relation)

As usual the step-index is needed to keep the value relation well-founded: the only argument of the relation that is structurally smaller than the original arguments in the recursive call of $\preceq_i$ is the step-index $i$. In turn, this makes it necessary to decrease the step-index by the amount of beta reductions that happen in the term $e_1$ when we call the value relation from the expression $e_1$. However, we must also show that $\lambda x.e_1 \preceq_i \lambda y.e_2$ when $e \rightarrow e'$. In this case, we do not need to decrease the step-index and we can define a simple step-indexed expression relation $\preceq_i$ that relates two expressions at step index $i$ if whenever the first expression reduces to some value $v_1$ with some number of steps $c_1 \leq i$, then the second expression also reduces to some value $v_2$ and the two values are related by the expression relation for the remaining $i - c_1$ steps. The expression relation $\preceq_i$ asserts that two lambda abstractions are related if they map any two arguments related at strictly smaller step indices to related results.

$$
\begin{align*}
\lambda x.e_1 \preceq_i \lambda y.e_2 & \iff \forall v_1, v_2, j (j < i), v_1 \preceq_j v_2 \Rightarrow e_1[v_1/x] \preceq_j e_2[v_2/y]
\end{align*}
$$

(Expression relation)
relation. Otherwise, we wouldn’t be able to prove that applications of related functions to related arguments are also related, i.e. if \((\lambda x.e_1) \leq_i (\lambda y.e'_1)\) and \(e_2 \leq_i e'_2\), then \((\lambda x.e_1)\ e_2 \leq_i (\lambda y.e'_1)\ e'_2\).

We define \(e_1 \leq e_2 \overset{\text{def}}{=} \forall i, e_1 \leq_i e_2\) and \(v_1 \leq v_2 \overset{\text{def}}{=} \forall i, v_1 \leq_i v_2\). We define a substitution relation \(S \vdash \sigma \leq_j \sigma'\) that holds when for any \(x \in S\), if \(\sigma(x)\) is defined then so is \(\sigma'(x)\) and furthermore \(\sigma(x) \leq_j \sigma'(x)\).

The logical relation has some useful properties that we will use in the correctness proof. First, it satisfies the usual compatibility property that asserts that two terms that are applications are logically related if their subterms are logically related.

\[
e_1 \leq_i e'_1 \Rightarrow e_2 \leq_i e'_2 \Rightarrow e_1 e_2 \leq_i e'_1 e'_2
\]

(Application compatibility)

Another property, which is key to our reasoning, is that the logical relation is a preorder:

\[
e \leq e
\]

(Reflexivity)

\[
e_1 \leq_i e_2 \Rightarrow e_2 \leq e_3 \Rightarrow e_1 \leq_i e_3
\]

(Transitivity)

Both of these properties are proved by induction on the step index simultaneously with the same properties of the value relation. Observe that the second premise in transitivity requires that expressions are related for all step-indices. To understand this consider the transitivity proof for expressions. Let \(e_1 \leq_i e_2\) (H1) and \(e_2 \leq e_3\) (H2). We need to show that \(e_1 \leq e_3\). Let \(e_1 \rightarrow^{c_1} v_1\) for some \(c_1 \leq i\) and \(v_1\). By H1, we obtain \(v_2\) and \(c_2\) such that \(e_2 \rightarrow^{c_2} v_2\) and \(v_1 \leq_i e_{-c_1} v_2\). We then instantiate H2 with the step index \(i - c_1 + c_2\) and \(e_2 \rightarrow^{c_2} v_2\). We obtain \(v_3\) and \(c_3\) such that \(e_3 \rightarrow^{c_1} v_3\) and \(v_2 \leq_{i-c_1} e_{-c_1} v_3\), or, equivalently \(v_2 \leq_{i-c_1} v_3\). Using the induction hypothesis on values, we obtain \(v_1 \leq_{i-c_1} v_1\) which concludes the proof. If \(e_2 \leq e_3\) were not related for all step indices, we would not be able to instantiate this premise with step index \(i - c_1 + c_2\).

Lastly, we can use the logical relation to relate a term before and after beta-reduction:

\[
e_1[v_2/x] \leq (\lambda x.e_1) v_2
\]

(Reduce-App)

We could show the other direction too, but for our proof this is enough. This property is key to our reasoning because we can use it in combination with transitivity to perform reductions on a term and end up with a term that is related to the original one. This gives us the flexibility of a small-step semantics without actually using one, avoiding the issues that arise because of administrative reductions.

To prove the CPS transformation correct, we need to prove that CPS transforming the same term using different (fresh) names for binders yields semantically the same terms. That is, the term \([e]_k^x\), which picks fresh variables names starting from \(x\), is semantically related to \([e]_k^y\), which picks fresh variables names starting from \(y\), as long as \(x\) and \(y\) are fresh with respect to \(e\) and \(k\). By using this lemma, we avoid making \(\alpha\)–conversion explicit in our theorem statement. First, we need to prove a more general fact.

**Lemma 2.1.**

1. For all terms \(e, k, k'\), substitutions \(\sigma\) and \(\sigma'\) such that \(\text{fv}(e) \subseteq \text{dom}(\sigma)\) and \(\text{fv}(e) \vdash \sigma \leq \sigma', \text{ x fresh w.r.t. } e \text{ and } k\), and \(y\) fresh w.r.t. \(e\) and \(k'\), we have that if \(\sigma(k) \leq \sigma'(k')\) then \(\sigma([e]_k^x) \leq \sigma'([e]_{k'}^y)\).

2. For all values \(v\), substitutions \(\sigma\) and \(\sigma'\) such that \(\text{fv}(v) \subseteq \text{dom}(\sigma)\) and \(\text{fv}(v) \vdash \sigma \leq \sigma', \text{ and } x, y\) that are fresh w.r.t. \(v\), we have that \(\sigma([v]_x^x) \leq \sigma'([v]_y^y)\).

**Proof.** We prove the statements simultaneously by induction on the step-index. Both statements follow easily from the induction hypothesis, the definition of the value relation, and application compatibility. \qed
The above lemma generalizes the fact that wish to prove by using the closing substitutions σ and σ′ to account for free variables. It also accounts for different but semantically related continuation terms. From the above lemma, we derive two useful corollaries: (1) \([e]_k^x \preceq [e]_k^y\) (name invariance), and (2) if \(k \leq k′\) then \([e]_k^x \preceq [e]_{k′}^x\) (related continuations). As is common in CPS proofs, we will also need some substitution properties. (1) We show that the CPS transformation commutes with substitution, up to the logical relation: \([e]_k^x[(\nu)\!^y/\nu]\) \(\preceq [e[\nu/z]]_{k[\nu/x]}^x\), for \(x\) fresh w.r.t. \(e, \nu, k,\) and \(z,\) and \(y\) fresh w.r.t. \(\nu.\) (2) We show that \([e]_k^x[\nu/z] \preceq [e]_{k[v/x]}^i\) (respectively \((\nu)\!^y[\nu'//\nu/z] \preceq (\nu)\!^y\) for values) for any \(z\) that does not appear free in \(e\) (resp. \(\nu\)).

With this machinery in hand, we can now state and prove the correctness theorem of the CPS transformation.

**Theorem 2.2** (Correctness of CPS). For any closed terms \(e\) and \(k,\) and \(x\) fresh w.r.t. \(e\) and \(k,\) we have that: \(e \rightarrow e′ \Rightarrow [e]_k^x \preceq [e']_k^x\)

**Proof.** We proceed by induction on the reduction relation. There are three cases to consider.

- **Case 1:** \(e_1 \rightarrow e_1′\)
  \(e_1 \rightarrow e_2 \rightarrow e_1′e_2\)

  We need to show that \([e_1′e_2]_k^x \preceq [e_1e_2]_k^x\). We have that \([e_1e_2]_k^x = [e_1]_k^y[λm.e_2]_{λn.m}[\nu/m] [\nu/n] + [e_2]_k^{x_12} \) and \([e_1′e_2]_k^x = [e_1′]_k^y[λm.e_2]_{λn.m}[\nu/m] [\nu/n] + [e_2]_k^{x_12} \), where \(y\) is the name generator after the call to \([e_2]_k^{x_12} \) for \(m\) and \(n\) are both \(x\). From the induction hypothesis we have that \([e_1]_k^y \preceq [e_1′]_k^y\).

- **Case 2:** \(e_2 \rightarrow e_2′\)
  \(v_1 \rightarrow v_2 \rightarrow v_1e_2\)

  We need to show that \([v_1e_2]_k^x \preceq [v_1′e_2′]_k^x\). We have that \([v_1e_2]_k^x = [v_1]_k^y[λm.e_2]_{λn.m}[\nu/m] [\nu/n] + [e_2]_k^{x_12} \) and \([v_1′e_2′]_k^x = [v_1′]_k^y[λm.e_2]_{λn.m}[\nu/m] [\nu/n] + [e_2]_k^{x_12} \), where \(y\) is the name generator after the call to \([e_2]_k^{x_12} \) for \(m\) and \(n\) are both \(x + 1\). Note that both \(y\) and \(y′\) are fresh w.r.t. to \(v_1\) and the two continuation terms. It suffices to show that \([λm.[e_2]_k^{x_2}]_{λn.m}[\nu/m′] [\nu/n′] \preceq [λm.[e_2]_k^{x_2}]_{λn.m}[\nu/m′] [\nu/n′] \) which is obtained from the induction hypothesis, using the definition of the logical relation, the substitution property (2), and the related continuations lemma that are combined using transitivity.

- **Case 3:** \((λz.e) \rightarrow e[\nu/z]\)

  We need to show that \([e[\nu/z]]_k^x \preceq [(λz.e)[\nu]]_k^x\). We have that \([(λz.e)[\nu]]_k^x = [λz.e]_{λm.[\nu]}_{λn.m}[\nu/m] [\nu/n] \) where \(y\) is the name generator after the call to \([\nu]_k^{x_2} \) for \(m\) and \(n\) are both \(x + 1\). From translation definition, we have \([λz.e]_{λm.[\nu]}_{λn.m}[\nu/m] [\nu/n] = (λm.(λn.m[k]) (\nu)_{x_2}) \) (\(λk′.λz,[e]_{k′}^{y+1}\)), where \(k′ = y\). We observe that:

  \(1\) \((λm.(λn.m[k]) (\nu)_{x_2}) \preceq (λn.(λk′.λz,[e]_{k′}^{y+1})) [k] (\nu)_{x_2} \)

  \(2\) \((λk′.λz,[e]_{k′}^{y+1}) k (\nu)_{x_2} \preceq (λz.[e]_{k′}^{y+1}) (\nu)_{x_2} \preceq [e]_{k′}^{y+1}[(\nu)_{x_2}/z] \)

  \(3\) \([e[\nu/z]]_k^{y+1} \preceq [e[\nu/z]]_k^x\)

  The above steps are obtained as follows.
We have established that our CPS transformation preserves the behaviors of terminating source programs. We will demonstrate how it is possible to show this fact by using a logical relation that is parameterized by fresh w.r.t. to $e[v/z]$ and $k$.

When we apply the induction hypothesis in cases 1 and 2, we have to discharge the closedness and freshness assumptions. These follow easily from the fact that substitution and reduction preserve closedness and binding names.

Using the above proof, we can derive a top-level corollary that ensures semantic preservation. If $e \rightarrow^* v$ then, using the transitivity property of the logical relation, we derive $[v]_{id}^x \leq [e]_{id}^x$, where $id$ is the identity function, used as the continuation for top-level terms. But $[v]_{id}^x = id \langle v \rangle^x$ which reduces to $\langle v \rangle^x$. Therefore, from the definition of the logical relation, there exists some value $v'$ such that $[v]_{id}^x \rightarrow^* v'$ and $\langle v \rangle^x \leq v'$. As we will demonstrate in the rest of the paper, this is enough to establish that the CPS transformation preserves the behavior of terminating source programs.

Let us now reflect on what we have achieved. By using a logical relation, we demonstrated that we can stick to a named binder representation (that makes implementation and some aspects of reasoning easier) and yet avoid the need to explicitly account for $\alpha$-equivalence. We also avoid using a colon translation or parallel reduction in order to state the simulation relation. The shape of our simulation relation and our proof resembles a simple small-step syntactic simulation. We conjecture that with this proof technique, Dargaye and Leroy’s one-pass CPS translation can be directly proved correct as their transformation commutes with substitution up to the logical relation.

However, he have not yet shown that CPS preserve diverging behaviors. In the next section, we will demonstrate how it is possible to show this fact by using a logical relation that is parameterized by resource invariants. Using the same mechanism we also show that our CPS transformation preserves the running time of source programs.

### 3 Reasoning about Running Time

We have established that our CPS transformation preserves the behaviors of terminating source programs. But correct transformations are also expected to preserve diverging behaviors, meaning that if the execution of the source program diverges then the execution of the target program should diverge too. An other important property for transformations is the preservation of the asymptotic running time of the source programs. That is, the number of steps needed to evaluate the target program should be within a constant factor of the steps required to evaluate the source program. These two properties have something in common: they require reasoning about the number of steps that programs take. In particular, by establishing a suitable relation between the number of steps of the source and target programs we can show both of these properties. We will do that by extending the logical relation with a resource invariant: a relation that is enforced on the steps of source and target programs. We start by restating the top-level correctness corollary of the CPS transformation to take into account preservation of running time and divergence.

**Diverging programs.** In order to prove divergence preservation, we first need a notion for diverging programs. We say that a program diverges if for any natural number $n$, there is a reduction sequence of length $n$ starting from this term, i.e., $e \Downarrow n \defeq \forall n, \exists e', e \rightarrow^n e'$.

Obviously, if we show that the target program takes at least as many steps as the source program then diverging behaviors are preserved.
Correctness. The extended top-level corollary for CPS consists of two facts.

1. If \( e \xrightarrow{c_1} v \) then there exists \( v' \) such that \( [e]^x_{id} \xrightarrow{c_1} v' \), \( [v]^y \leq v' \), and \( c_2 \leq A + c_1 + B \), where \( A \) and \( B \) are constants.

2. If \( e \Downarrow \) then \( [e]^x_{id} \Downarrow \).

The new logical relation that we will define let us establish both of these facts with only minor adjustments to the proof of the theorem 2.2.

3.1 Extending the Logical Relation

To relate the steps of the source and target programs, we parameterize the expression logical relation with two relations, the local invariant and the global invariant. We write \( e_1 \xpreceq_i^{Q_L, Q_G} e_2 \) to denote that expressions \( e_1 \) and \( e_2 \) are logically related at the step index \( i \) under the local invariant \( Q_L \) and the global invariant \( Q_G \). Similarly, we write \( v_1 \xpreceq_i^{Q_G} v_2 \) to denote that the values \( v_1 \) and \( v_2 \) are related at step index \( i \) under the global invariant \( Q_G \). The new definitions are shown below.

\[
e_1 \xpreceq_i^{Q_L, Q_G} e_2 \overset{def}{=} \forall \: c_1. \: e_1', \: c_1 \leq i \Rightarrow e_1 \xrightarrow{c_1} e_1' \Rightarrow \exists \: e_2', \: e_2 \xrightarrow{c_2} e_2' \land \left( \text{value}(e_1') \Rightarrow e_1' \xpreceq_i^{Q_G} e_2' \right) \land Q_L(c_1, c_2)
\]

\[
\lambda x. e_1 \xpreceq_i^{Q_G} \lambda y. e_2 \overset{def}{=} \forall \: v_1. \: v_2 \quad (j < i), \quad v_1 \xpreceq_j v_2 \Rightarrow e_1[v_1/x] \xpreceq_j^{Q_G, Q_G} e_2[v_2/y]
\]

The expression relation is different that the previous definition of the logical relation in that it additionally enforces that the steps of the source program \( c_1 \) and the steps of the target program \( c_2 \) are related by the local invariant \( Q \). The global invariant is used as a parameter for the value relation. In turn, the value relation uses the global invariant to instantiate both the local invariant and the global invariant when it invokes the expression relation for function bodies. Conceptually, the local invariant is used to relate the steps of to expressions locally, at their current execution state. The global invariant is used to relate the execution steps of whole function bodies.

We first define shorthands for some useful invariants. We write \( m= \) for the invariant \( \text{fun} c_1 c_2 \Rightarrow c_1 + m = c_2 \). Similarly, we write \( m\leq \) (resp. \( m\geq \)) for the invariant \( \text{fun} c_1 c_2 \Rightarrow c_1 + m \leq c_2 \) (resp. \( \text{fun} c_1 c_2 \Rightarrow c_1 + m \geq c_2 \)). We write \( Q_1 \cap Q_2 \) to denote the conjunction of two invariants \( Q_1 \) and \( Q_2 \).

We highlight the distinction between the local and the global invariant with an example. We consider the expressions \( e_1[v_2/x] \) and \( (\lambda x. \ e_1) \ v_2 \). The two expressions compute exactly the same result, so they ought to be semantically related. Moreover, applications of related functions in the two results will evaluate using exactly the same number of reductions since their bodies are equal. So the global invariant is \( 0= \). But the two expression do not evaluate using the same number of reductions. If the expression \( e_1[v_2/x] \) evaluates using \( c \) reductions then the expression \( (\lambda x. \ e_1) \ v_2 \) evaluates using \( c + 1 \) reductions. The local invariant for which the two expressions are related is \( 1= \). Therefore, we have that \( e_1[v_2/x] \xpreceq_1^{(1=, 1=)} (\lambda x. \ e_1) \ v_2 \) (Reduce-App).

Properties of the logical relation. We now revisit the properties of the logical relation extended with resource invariants. We say that an invariant \( Q \) is compatible if whenever \( Q \ (x, y) \) then \( Q \ (x + 1, y + 1) \). This is a useful property because it entails that when an invariant holds for two programs, then it will continue to hold when both programs take a step.
The expression logical relation is monotonic in the local invariant. If two expressions are related for \( Q \), then they will also be related for any \( Q' \) that is weaker than \( Q \).

\[ Q \supseteq Q' \Rightarrow e_1 \preceq_{\{Q, Q_G\}} e_2 \Rightarrow e_1 \preceq_{\{Q', Q_G\}} e_2 \quad \text{(Local invariant monotonicity)} \]

Note that the relation is not monotonic in the global invariant, as it appears in both negative and positive positions in the definition of the value relation.

For the extended logical relation, reflexivity holds as long as both the local and global invariant are compatible. Therefore, for compatible \( Q \) and \( Q_G \), we have \( e \preceq_{\{Q, Q_G\}} e \) (Reflexivity).

To prove transitivity, we need to compose the local invariants of the two premises. That is, if expressions \( e_1 \) and \( e_2 \) are related with \( Q_1 \), and expressions \( e_2 \) and \( e_3 \) are related with \( Q_2 \), then \( e_1 \) and \( e_3 \) will be related with \( Q_1 \circ Q_2 \). However this is not the case for the global invariant, that needs to be the same in the two premises and the result. For this reason, we also need to assume that the global invariant \( Q_G \) is semi-idempotent, i.e. \( Q_G \circ Q_G \Rightarrow Q_G \). This is useful when we use the induction hypothesis when we prove transitivity for the value relation. Lastly, because the proof of transitivity of the value relation uses reflexivity, we need to assume that \( Q_G \) is compatible. Given these assumptions on the relational invariants, we can prove transitivity.

\[ e_1 \preceq_{\{Q, Q_G\}} e_2 \Rightarrow e_2 \preceq_{\{Q, Q_G\}} e_3 \Rightarrow e_1 \preceq_{\{Q, Q_G\}} e_3 \quad \text{(Transitivity)} \]

**Proof of CPS.** Using this logical relation we can prove a stronger theorem about CPS. First we need to adjust lemma 2.1, which states that CPS produces semantically related terms when instantiated with different fresh name generators, by picking the right local and global invariants. Because all related expressions are identical up to \( \alpha \)-conversion, the number of reduction steps remains the same. Therefore, all invariants that are \( 0= \) (we do not repeat the lemma here). As before, we derive that (1) \( \|e\|_k^x \leq^{(n, n, 0)} \|e\|_k^0 \) (name invariance), and (2) if \( k \leq^{(n, n, 0)} k' \) then \( \|e\|_k^x \leq^{(n, n, 0)} \|e\|_{k'}^0 \) (related continuations). Substitution properties also need a similar adjustment. Again, all invariants are \( 0= \) and we do not repeat the properties.

In order to state and prove the new CPS theorem, we need to find suitable resource invariants. Given \( e \rightarrow e' \), we need to relate the terms \( \|e'\|_k^x \) with \( \|e\|_k^x \). These two terms compute the same result up to \( \alpha \)-conversion, so the choice for the global invariant is obvious: as before, we pick \( 0= \). The choice of the local invariant is trickier. Recall that we need both an upper bound and a lower bound for the number of reductions in the target term. The lower bound is used to prove divergence preservation, while the upper bound to prove preservation or running time. For the lower bound, consider that reduction sequences in the term \( \|e\|_k^x \) must always be strictly less than the reductions in the term \( \|e\|_k^x \), since at least one reduction has been performed. This is captured by the invariant \( 1\leq \). To find the the upper bound, let \( c \) be the reductions needed to evaluate the term \( \|e\|_k^x \). The number of reduction needed to evaluate the term \( \|e\|_k^x \) are \( c + 4 \). This is because a \( \beta \)-reduction in the source programs gives rise to three additional \( \beta \)-reductions in the target. This will become clearer during the proof.

**Invariant composition.** The invariants that we defined earlier have some useful compositionality properties. In particular we have that:

1. \( n = \circ m \geq n + m = \)
2. \( n \leq \circ m \leq n + m \leq \)
3. \( n \geq \circ m \geq n + m \geq \)

We will use these during the proof to compose the invariants of subsequent transitivity steps.

The new theorem for CPS is the following.

**Theorem 3.1 (Correctness of CPS, revisited).** For any closed terms \( e \) and \( k \), \( i \) fresh w.r.t. \( e \) and \( k \), we have that: \( e \rightarrow e' \Rightarrow \|e'\|_k^x \leq^{(n, n, 0)} \|e\|_k^x \)
Before outlining the proof of this theorem, let us explain how from this theorem we can derive the top-level corollary that we wish to prove. First, we demonstrate how we derive divergence preservation. Let \( e \parallel \). We need to show that \([e]_{id}^{0} \parallel\). For any \( n \), we will show that there exists \( e' \), such that \([e]_{id}^{n} \rightarrow e'\). From the hypothesis that \( e \) diverges, we have that there exists \( e'' \) such that \( e \rightarrow^{n} e''\). By subsequent applications of theorem 3.1, the transitivity lemma, and the compositionality properties of the invariants, we derive \([e''']_{id}^{x} \leq ^{(n+1)_{id} \geq 0} [e]_{id}^{x}\). We can instantiate this statement with \([e''']_{id}^{n} \rightarrow^{0} [e']_{id}^{x}\), and obtain \( e'''' \) and \( n' \) such that \([e']_{id}^{n} \rightarrow^{n'} e''''\) and \( 0 + n \leq n' \) (by the resource invariant \( ^{0}\)). Therefore, there also exists \( e' \) such that \([e]_{id}^{x} \rightarrow^{n} e'\) which concludes the proof.

For preservation of terminating behaviors, let \( e \rightarrow^{n} v \). Again, by subsequent applications of theorem 3.1, the transitivity lemma, and the compositionality properties of the invariants, we derive \([v]_{id}^{x} \leq ^{(n+1)_{id} \geq 0} [v]_{id}^{x}\). We have that \([v]_{id}^{x} = \mathit{id}(v)^x\). By lemma reduce-App, we have \([v]_{id}^{x} \leq ^{(n+1)_{id} \geq 0} \mathit{id}(v)^x\). Then, by one more application of transitivity, we obtain \([v]_{id}^{x} \leq ^{(n+1)_{id} \geq 0} [v]_{id}^{x}\). Because \([v]_{id}^{x} \rightarrow^{0} [v]_{id}^{x}\) and \([v]_{id}^{x}\) is a value, we obtain \( v' \) and \( n' \) such that \([e]_{id}^{x} \rightarrow^{n'} v'\), \([v]_{id}^{x} \leq ^{0\sim} v'\), and \( n' \leq 4 * n + 1 \) (by the invariant \( ^{4}\)).

We now return to the proof of theorem 3.1. The proof remains largely the same with the proof of theorem 2.2. The only difference is the addition of resource invariants.

**Proof.** To demonstrate the handling of resource invariants, we outline the proof of the third case.

- **Case 3:** \((\lambda z.e) \ u \rightarrow e[v/z]\)
  
  We need to show that \([e[v/z]]_{k}^{x} \leq ^{(n+1)_{id} \geq 0} v\). We have that \([v]_{id}^{x} = \mathit{id}(v)^x\). We have \([v]_{id}^{x} \leq ^{(n+1)_{id} \geq 0} [v]_{id}^{x}\). Because \([v]_{id}^{x} \rightarrow^{0} [v]_{id}^{x}\) and \([v]_{id}^{x}\) is a value, we obtain \( v' \) and \( n' \) such that \([e]_{id}^{x} \rightarrow^{n'} v'\), \([v]_{id}^{x} \leq ^{0\sim} v'\), and \( n' \leq 4 * n + 1 \) (by the invariant \( ^{4}\)).

4 **Mechanization Setup: Source and Target Languages**

We now move to our actual mechanization setting, which is part of the CertiCoq compiler. One of the main differences is that CertiCoq uses big-step semantics in its correctness proofs throughout the pipeline. In the rest of the paper, we demonstrate how our novel proof technique can be adapted to big-step semantics, while still supporting divergence preservation. We have made our mechanization publicly available in an open-access repository [Paraskevopoulou and Grover 2021].
The source language we consider, $\lambda_{DS}$, is a direct-style lambda calculus extended with let binding, (mutually)$^2$ recursive functions, constructors, and pattern matching. Binders are represented with DeBruijn indices. Our CPS translation moves from De Bruijn representation, which is used in the front-end of the CertiCoq compiler, to a named binder representation, which is used in the rest of the compiler pipeline. The syntax of the language (in sans serif font) is shown in fig. 1. The constructs in order they appear are: variable, let-binding, (recursive) function definition, function application, constructor, and pattern matching. The let construct binds one variable in the scope of $e_2$ (which holds the value of $e_1$), the fix construct binds two variables (the argument and the function itself) in $e$, and the match construct binds $m_i$ variables in $e_i$, where $m_i$ is the number of constructor arguments of $C_i$. Values in $\lambda_{DS}$ are either constructed values or closures. Closures are represented as pairs of an environment and a function definition. The environment is represented as a list of variables, with the element in the $i$-th position corresponding to the $i$-th De Bruijn index.

$$e ::= \text{let } x = C(y) \text{ in } e \quad | \quad \text{let } x = y.i \text{ in } e \quad | \quad \text{fun } f \bar{x} = e_1 \text{ in } e_2 \quad | \quad \text{case } y \text{ of } [C_i \rightarrow e_i]_{i \in I} \quad | \quad f \bar{x} \quad | \quad \text{ret}(x)$$

$$v ::= C(\bar{v}) \quad | \quad \text{Clos}(\sigma, \text{fun } f \bar{x} = e)$$

$$\sigma = \text{Var} \rightarrow \text{Val}$$

Fig. 1. Source language ($\lambda_{DS}$).

The target language, $\lambda_{CPS}$, is shown in fig. 2.$^3$ This language uses named binders. It is a low-level language that requires all intermediate results to be fully evaluated. The result of each (non-tail) operation is explicitly bound to a variable. The constructs in the order they appear are: constructors, projections, case analysis, (mutually) recursive function definitions, tail calls and function returns (that are used only by the top-level continuation). In $\lambda_{CPS}$, the patterns of a case analysis statement do not bind any variables, they simply discriminate the outermost constructor of a constructed value. The CPS translation generates explicit projections that project and bind the arguments of the constructor in each branch expression. The values of the language are either constructed values or closures. Closure environments are represented as partial maps from variable names to values. Observe that unlike $\lambda_{DS}$, where all abstractions and applications are unary, $\lambda_{CPS}$ has multi-argument functions. Our CPS translation uses two arguments (original argument and continuation) for user-level functions and one argument for continuations. The pipeline of CertiCoq introduces multi-argument functions with an uncurrying transformation after CPS/ANF transformation.

$^2$We omit mutually recursive function from the paper presentation of the source and target languages. Our implementation and mechanization fully support mutually recursive functions.

$^3$Our target language is actually an A-normal form representation, $\lambda_{ANF}$, which the core intermediate representation of the CertiCoq compiler. $\lambda_{ANF}$ is a superset of $\lambda_{CPS}$ is that has an additional construct for calls that are not tail recursive. CertiCoq supports both direct-style and CPS compilation by using either a CPS or ANF transformation between $\lambda_{DS}$ and $\lambda_{ANF}$. Because the CPS translation does not generate calls that are not tail recursive, here we restrict our target language to $\lambda_{CPS}$.
4.1 Semantics

For both source and target languages, we use environment-based, fuel-based semantics, as the rest of the CertiCoq pipeline. The fuel tracks the steps needed to evaluate a program and it is used to establish divergence preservation, as in other compilers [Owens et al. 2016, 2017]. The result of evaluation in both the source and the target languages is either a value (wrapped in the Res() constructor) or an out-of-time exception (OOT), which is thrown when evaluation has run out of fuel.

\[ r := \text{Res}(v) \mid \text{OOT} \]

**Source language.** The evaluation judgment of the source language, written \( \mathcal{U}; e \Downarrow^{(f,c)}_{sr} r \), asserts that a source configuration, that consists of an environment \( \mathcal{U} \) (represented as a list of values) and a term \( e \), evaluates in \( f \) steps to some result \( r \). The result may be either a \( \lambda_{DS} \) value or an out-of-time exception. Each evaluation rule consumes a unit of fuel. The fuel value \( f \) is essentially the height of the evaluation derivation. The evaluation also keeps track of the cost of the program, \( c \), which accounts for the cost of evaluating each language construct after CPS conversion. We will use the cost to give an upper bound to the evaluation steps needed for the CPS-converted program. The evaluation rules are mostly standard and shown in fig. 3. When a rule requires that a subexpression evaluates to a value we need to consider two cases: either the amount of fuel is enough to fully evaluate the subexpression, or the fuel is not enough for the evaluation of the subexpression, therefore we propagate the out-of-time exception. For example, we have three rules for application. The rule \( \text{App} \) asserts that an application \( e_1 e_2 \) evaluates to a result \( r \) in some amount of fuel \( f \) if (1) \( e_1 \) evaluates to a closure using fuel \( f_1 \), (2) \( e_2 \) evaluates to to a value using fuel \( f_2 \), (3) the body of the function in its closure environment extended with the argument and the closure yields a result using fuel \( f_3 \), and (4) \( f \) is equal to \( f_1 + f_2 + f_3 + 1 \), where the unit accounts for the cost of evaluating the application. Rule \( \text{App-OOT}_1 \) asserts that evaluating the application throws an out-of-time exception for any amount of fuel that is not enough to evaluate \( e_1 \). Rule \( \text{App-OOT}_2 \) asserts that evaluating an application throws an out-of-time exception for any amount of fuel that is not enough to evaluate both \( e_1 \) and \( e_2 \). Similarly, there are two cases for evaluating constructors. The first rule (\( \text{Constr} \)) asserts that whenever all constructor arguments evaluate to a value, then the constructor evaluates to the constructor value. The second rule (\( \text{Constr-OOT} \)) asserts that the evaluation of a constructor times out for any amount of fuel that is not enough to evaluate all of its arguments. Rule OOT throws an out-of-time exception when the fuel runs out.

The cost of incurred by each rule is the cost required for evaluating the CPS translation of each construct. This is a constant overhead for most rules, except from constructors and pattern matching where the cost depends on the size of the program. The cost for constructors depends on the number of the constructor arguments (because the translation defines a continuation for each argument). The cost for pattern matching depends on the (maximum) number of constructor arguments because the translation will explicitly project such argument in each branch.

Just as closure environments, the evaluation environment in this semantics is simply a list of values. The value with index \( i \) in the environment is the value of the free De Bruijn variable \( i \). When a binder is “opened” and instantiated with a value, this value is pushed to the beginning of the list. For example, in the rule for pattern matching (\( \text{Match} \)), in order to evaluate the expression of the matching branch in which the arguments of the constructor are bound, we push the values of the arguments of the constructor to the environment. Notice that the order needs to be reversed since the head of the environment list holds the value of the most recently bound variable.

\(^4\) Even though CertiCoq is used to compile a terminating language, its proof aims to be more general to accommodate compilation of non-terminating languages in the future.
\[
\begin{align*}
\vdash f \text{ from } \sigma & \mid \mathbf{Res}(f) \\
\langle v_1, \ldots, v_n, \ldots, v_m; n \rangle & \vdash_{\text{src}}^{(1,1)} \mathbf{Res}(v_n) & \text{VAR} \\
\langle v \rangle & \vdash_{\text{src}} \mathbf{Res}(\mathbf{fix}(v)) & \text{Fix} \\
\langle e \rangle & \vdash_{\text{src}} \mathbf{Res}(\mathbf{Clos}(\mathbf{fix}(e))) & \mathbf{Clos}(\mathbf{fix}(e)) \\
\langle v \rangle & \vdash_{\text{src}} \mathbf{Res}(\mathbf{App}(v_1, \ldots, v_n)) & \mathbf{App} \\
\langle f, e \rangle & \vdash_{\text{src}} \mathbf{OOT}(f+1, c+5) & \mathbf{App-OOT_1} \\
\langle v \rangle & \vdash_{\text{src}} \mathbf{Res}(\mathbf{C(n, \ldots, v_n)}) & \mathbf{Res}(\mathbf{C(n, \ldots, v_n)}) \\
\langle f, e \rangle & \vdash_{\text{src}} \mathbf{Clos}(\mathbf{fix}(e)) & \mathbf{Clos}(\mathbf{fix}(e)) \\
\langle f, c \rangle & \vdash_{\text{src}} \mathbf{Res}(\mathbf{App}(v_1, \ldots, v_n)) & \mathbf{App-OOT_2} \\
\langle v \rangle & \vdash_{\text{src}} \mathbf{Res}(\mathbf{Constr}(v_1, \ldots, v_n)) & \mathbf{Constr} \\
\langle f, e \rangle & \vdash_{\text{src}} \mathbf{Constr-OOT}(f+1, c+5) & \mathbf{Constr-OOT} \\
\langle v \rangle & \vdash_{\text{src}} \mathbf{Match}(e) & \mathbf{Match} \\
\langle f, e \rangle & \vdash_{\text{src}} \mathbf{Match-OOT}(f+1, c+3) & \mathbf{Match-OOT} \\
\langle \sigma, e \rangle & \vdash_{\text{trg}} (0, 0) & \mathbf{Out} \\
\end{align*}
\]

Fig. 3. Semantics of \(\lambda_\text{DS}\) (excerpt). We omit the two rules for let that are straightforward.

**Target language.** The semantics of the target language (fig. 4) are written in a similar spirit. The evaluation judgment is written \( (\sigma, e) \vdash_{\text{erg}} f \), where \( \sigma \) is the environment (a partial map from variables to values), \( e \) the expression, \( f \) the final result and \( r \) the fuel value. Because of the linear nature of CPS representation, no rule requires the evaluation of more than one expression in its premises. Therefore, the semantics is less verbose as we do not need to distinguish cases for the evaluation of subexpressions. The evaluation rules are unsurprising. The rule for constructor (resp. projection), evaluates the constructor (resp. projection) and binds it to the variable of the let-binding before evaluating the rest of the program. The same is true of the rule for evaluating a function definition (FUN). A closure is constructed by pairing the current environment with the function definition and is then added to the environment before proceeding with the evaluation. The rule for case analysis (CASE) simply evaluates the expression of the branch that matches the contractor tag. Since the pattern expression in the target does not bind any variables, the environment is not extended. The rule for function application (App) fetches the value of the closure and the arguments from the environment and evaluates the body of the function in the closure environment extended with the appropriate bindings for the function name and the arguments. The rule for return (Ret) simply returns the value of a variable.
\[
\begin{align*}
\sigma(\vec{y}) = \vec{v} & \quad (\sigma[x \mapsto C(\vec{v})], e) \trianglelefteq_{\text{trg}}^{c} r \quad \text{CONSTR} \\
(\sigma, \text{let } x = C(\vec{y}) \text{ in } e) & \trianglelefteq_{\text{trg}}^{c+1} r \\
(\sigma, \text{let } x = C(\vec{v}), y \mapsto v_j \text{ in } e) & \trianglelefteq_{\text{trg}}^{c+1} r \quad \text{PROJ} \\
\sigma(x) = C(v_1, \ldots, v_j, \ldots, v_n) & \quad (\sigma[x \mapsto v_j], e) \trianglelefteq_{\text{trg}}^{c} r \quad \text{CASE} \\
(\sigma, \text{case } x \text{ of } \{C_i \rightarrow e\}_{i \in I}) & \trianglelefteq_{\text{trg}}^{c+1} r \\
(\sigma, f \vec{y}) & \trianglelefteq_{\text{trg}}^{c+1} r \\
\sigma(\vec{y}) = \vec{v} & \quad (\sigma_f, \text{fun } g \vec{z} = e_g) \\
(\sigma, e) & \trianglelefteq_{\text{trg}}^{c} r \quad \text{APP} \\
\sigma(\vec{y}) = \vec{v} & \quad (\sigma, \text{ret}(x)) \trianglelefteq_{\text{trg}}^{1} \text{ Res}(v) \quad \text{RET} \\
(\sigma, e) & \trianglelefteq_{\text{trg}}^{0} \quad \text{OOT} \\
\end{align*}
\]

Fig. 4. Semantics of \( \lambda_{\text{CPS}} \).

**Divergence.** We say that a program diverges if it throws an out-of-time exception for any possible fuel value. We define

\[
\exists f \in \text{src} \ni \forall c \in \text{src}, \quad \exists \sigma; e \trianglelefteq_{\text{src}}^{(f,c)} \text{ OOT}
\]

for the source language, and

\[
\exists f \in \text{trg} \ni \forall c \in \text{trg}, \quad (\sigma, e) \trianglelefteq_{\text{trg}}^{c} \text{ OOT}
\]

for the target language.

**Divergence preservation.** To show divergence preservation, it is enough to show that the fuel consumption of the source is always upper bounded by the fuel consumption of the target program. To prove this, we need a fuel monotonicity property for the target language: if a program times out with some amount of fuel, then it must do so for any smaller amount of fuel.

**Lemma 4.1 (Fuel monotonicity).** If \((\sigma, e) \trianglelefteq_{\text{trg}}^{f} \text{ OOT and } f' \leq f\) then \((\sigma, e) \trianglelefteq_{\text{trg}}^{f'} \text{ OOT.}\)

Then we can prove that if whenever the source times out with some amount of fuel the target also times out with at least the same amount of fuel, divergence is preserved.

**Lemma 4.2 (Divergence preservation).** Assume that whenever \(\exists f \in \text{src} \ni \sigma; e \trianglelefteq_{\text{src}}^{(f,c)} \text{ OOT we have that } (\sigma, e') \trianglelefteq_{\text{trg}}^{f} \text{ OOT for some } f' \geq f\). Then if \(\exists \sigma; e \trianglelefteq_{\text{src}}^{c} \text{ we have that } \sigma; e' \trianglelefteq_{\text{trg}}^{f} \text{ OOT.}\)

**Proof.** Let \(f\) be a fuel value. We need to show that \((\sigma, e') \trianglelefteq_{\text{trg}}^{f} \text{ OOT.}\) From the hypothesis that the source diverges, we have that \((\sigma, e) \trianglelefteq_{\text{src}}^{(f,c)} \text{ OOT for some } c\). Therefore, from the first hypothesis, we know that there exists \(f' \geq f\) such that \((\sigma, e') \trianglelefteq_{\text{trg}}^{f'} \text{ OOT.}\) From lemma 4.1 we obtain \((\sigma, e') \trianglelefteq_{\text{trg}}^{f} \text{ OOT.}\) \(\square\)
This style of reasoning is employed by both the CakeML [Owens et al. 2016, 2017] and the CertiCoq [Paraskevopoulou and Appel 2019; Paraskevopoulou et al. 2021] compilers. CertiCoq generalizes the previous lemma to also apply to transformations that may reduce the amount of steps that a program takes.

**Fuel vs. cost.** In the semantics of the source program, the fuel is used to track the number of rules that are required for the evaluation of the source program, while the cost tracks the number of rules needed after CPS conversion. Tracking the cost separately allows us to avoid program-dependent bounds for the fuel of the CPS program. Recall that for constructors and projections the overhead of the CPS transformation depends on the size of the source program. If we wanted to upper bound the fuel consumption of the target with that of the source, then we would end up with an upper bound of the form \( f_{\text{trg}} \leq g(f_{\text{src}}, e_{\text{src}}) \) for some function \( g \), which is harder to establish. By encoding this overhead in the cost model of the source, we can instead prove the simpler relation \( f_{\text{trg}} \leq c_{\text{src}} \), where \( c_{\text{src}} \) is the cost of the source.

**Big-step vs. small-step semantics.** We use big-step semantics for the CPS proof because this is the approach followed by the rest of the CertiCoq compiler. As others have observed [Dargaye and Leroy 2007; Leroy 2009; Owens et al. 2016], big-step semantics lead to more compositional proofs (by induction on the evaluation derivation) and simpler simulation diagrams. As we demonstrated in section 2 and section 3, our novel proof technique works well with small-step semantics as it avoids commonly arising issues and leads to compositional proofs. But as we do not adapt all of the proofs of the compiler, we stick to big-step semantics so that our proof composes with those of the rest of the pipeline. Compared to the small-step approach, big-step semantics require that we keep track of the fuel in order to show divergence preservation. This leads to rule duplication in the source language as several cases need to be considered. Additionally, as we will see in section 7, we also need to distinguish two cases for the main CPS theorem: one for when the source terminates and one for when the source times out. Nevertheless, our proof technique can be fully adapted to big-step semantics and the main ideas remain the same as in our earlier sections.

5 CONTINUATION-PASSING STYLE TRANSLATION

We now present our translation to continuation-passing style. The translation is an extension of the translation that we used as an example in section 2. We implement the translation both as a program and as a relational specification. This allows us to perform the proof without worrying about the details of the implementation. We then prove that the implementation satisfies the relational specification. This proof is easy as the shape of the transformation closely resembles the shape of the relation and it does not require any ingenuity.

The relational specification is written \( S; e; \bar{x}; k \triangleright e'; S' \) and it is expressed as an inductive relation. The first argument \( S \) is the set from which we draw fresh variable names—recall that our transformation moves from De Bruijn indices to a standard named binder representation. Whenever the transformation needs to pick a binder name, a variable from the set is used and removed from the set. \( S' \) is the set of remaining free variables that is threaded through the recursive calls. The arguments \( e \) and \( e' \) are the source and target expressions respectively. The argument \( \bar{x} \) is the variable environment, the list of variable names that correspond to the names of the free De Bruijn indices in the source program. \( k \) is the current continuation variable.

The rules of the CPS relation are shown in fig. 5. The rule for variable (\( \text{VAR} \)) looks up the index of the De Bruijn index in the environment in order to find the corresponding target variable and applies the current continuation to the target variable. The rule for fix (\( \text{Fix} \)) first defines the CPS-converted function and then applies the current continuation to the function binding. The CPS-converted target function has two arguments: the continuation argument and the original one. The body of the function is CPS-converted with its continuation argument as the current continuation and
the environment extended with the variable names for the function and the argument bindings.
The rule for application (App) generates a term that will evaluate the CPS-conversion of \( e_1 \) with a
continuation that evaluates the CPS-conversion of \( e_2 \) with a continuation that applies the function to the
current continuation and its original argument. The rules for constructor and pattern matching
are the most cumbersome. The rule for constructor (Constr) will create \( m \) nested continuations, one
for each argument of the constructor. Each continuation defines the next continuation in its body
and evaluates the CPS-converted constructor argument. Lastly, the current continuation is applied
to the constructor whose arguments are the continuation arguments. The rule for pattern matching

\[
\frac{\bar{x}.n = z}{S; n; \bar{x}; k \triangleright k \triangleright z; S} \quad \text{VAR}
\]

\[
\frac{x \in S \quad f \in S \setminus \{x\} \quad k_1 \in S \setminus \{f, k_1\} \quad S \setminus \{x, f, k_1\}; e; x :: f :: \bar{x}; k_1 \triangleright e' \triangleright S'}{S; \text{fix}(e); \bar{x}; k \triangleright \text{fun } f \; k_1 \; x = e' \; \text{in } f \; k; S'} \quad \text{FIX}
\]

\[
\frac{x_1 \in S \quad k_1 \in S \setminus \{x_1\} \quad x_2 \in S \setminus \{x_1, k_1\} \quad S \setminus \{x_1, x_2, k_1\}; e_1; \bar{x}; k_1 \triangleright e'_1 \triangleright S' \quad S' \setminus \{x_1, x_2\}; e_2; \bar{x}; k_2 \triangleright e'_2 \triangleright S''}{S; e_1; e_2; \bar{x}; k \triangleright \text{fun } k_1 \; x_1 = \text{fun } k_2 \; x_2 = x_1 \; k \; x_2 \; \text{in } e'_2 \; \text{in } e'_1 \; S''} \quad \text{APP}
\]

\[
\frac{x_1 \in S \quad k_1 \in S \setminus \{x_1\} \quad \{x_1, \ldots, x_m\} \in S \setminus \{x\} \quad \forall i, j. i \neq j \Rightarrow x_i \neq x_j \quad S \setminus \{x, x_1, \ldots, x_m\}; e_1; \bar{x}; k_1 \triangleright e'_1 \triangleright S_1 \quad \ldots \quad S_{m-1}; e_m; \bar{x}; k_m \triangleright e'_m \triangleright S_m}{S; \text{let } e_1 \; \text{in } e_2; \bar{x}; k \triangleright \text{fun } k_1 \; x_1 = e'_2 \; \text{in } e'_1 \; S''} \quad \text{LET}
\]

\[
\frac{x \in S \quad \{x_1, \ldots, x_m\} \in S \setminus \{x\} \quad \forall i, j. i \neq j \Rightarrow x_i \neq x_j \quad S \setminus \{x_1, \ldots, x_m\}; e_1; \bar{x}; k \triangleright e'_1 \triangleright S_1 \quad \ldots \quad S_{m-1}; e_m; \bar{x}; k_m \triangleright e'_m \triangleright S_m}{\text{fun } k \; x_1 = \ldots \text{fun } k \; x_m = \text{let } x = C(x_1, \ldots, x_m) \; \text{in } k \; x \; \text{in } e'_m \; \text{in } S_m} \quad \text{CONSTR}
\]

\[
\frac{\text{match } e \text{ with } [C_i \rightarrow e_i]_{i \in [1, n]} \bar{x}; k \triangleright \text{fun } k \; y = \text{case } y \text{ of } [C_i \rightarrow e'_i]_{i \in [1, n]} \; \text{in } e' \; S''}{S; [ ]; []; \bar{x}; k \triangleright [ ]; S} \quad \text{MATCH}
\]

\[
\frac{\{x_1, \ldots, x_m\} \in S \quad \forall i, j. i \neq j \Rightarrow x_i \neq x_j \quad S \setminus \{x_1, \ldots, x_m\}; e_1; \bar{x}; k \triangleright e'_1 \triangleright S' \quad S'; \bar{e}; \bar{m}; \bar{x}; k \triangleright e' \triangleright S''}{S'; e :: \bar{e}; m :: \bar{m}; \bar{x}; k \triangleright \text{let } x_1 = y \cdot 1 \; \text{in } \ldots \; \text{let } x_m = y \cdot m \; \text{in } e' :: e'' \; S''} \quad \text{BRANCH-CONS}
\]
(MATCH) CPS-converts the scrutinee with a continuation that performs the case analysis on the argument of the continuation. The expressions for the branches of the case analysis statement are CPS-converted using the auxiliary judgment $S; \bar{e}; \bar{m}; k; x \triangleright \bar{e}'; S'$ that will CPS convert each expression and will generate projections for the arguments of the constructor which are no longer bound by the pattern. In this judgment, $S$ is the fresh set, $\bar{e}$ are the pattern expressions, $\bar{m}$ the number of arguments of each constructor, $k$ the current continuation, $x$ the scrutinee, $\bar{e}'$ the CPS-converted expressions and $S'$ the resulting free set.

$$\forall i \in [1, m], \sigma(x_i) = \nu_i' \land \nu_i \triangleright \nu_i' \quad \text{ENV} \quad \forall i \in [1, m], \nu_i \triangleright \nu_i' \quad \text{CONSTR}$$

$$\bar{x} \in S \setminus \{f, y, k\} \quad \text{NoDup(} \bar{x} \text{)} \quad x \in S \setminus \{f, y\} \quad y \in S \setminus \{f\} \quad k \in S \setminus \{f, y, \bar{x}\} \quad e; y :: f :: \bar{x}; k \triangleright \bar{e}'; S' \quad \text{CLOSER}$$

Fig. 6. CPS environment and value relations.

A useful property of the transformation, that we use in the correctness proof, is that the fresh set after the transformation is a subset of the original fresh set: $S; e; \bar{x}; k \triangleright \bar{e}'; S'$ then $S' \subseteq S$. Additionally, we prove that the relation is total on well-scoped input terms.

In order to state the CPS correctness theorem we need to define CPS relations on values and on environments. These are shown in fig. 6. The environment relation (rule ENV) is written $\bar{x}; \bar{v} \triangleright \sigma$ and relates a variable environment to a source value environment and a target environment. The relation holds whenever the target environment has bindings for all of the variables in the variable environment and the values of the variables in the two environments are related with the CPS value relation.

The value relation relates two constructors (rule CONSTR) if their arguments are pairwise related. It relates a source closure and a target closure (rule CLOSER), if there exists a fresh set and variable names for the function name, the function argument, the continuation, and the variable environment, all distinct from each other, such that (1) the variable environment, the source closure environment, and the target environment are related with the CPS environment relation, and (2) the bodies of the function parts of the closures are related with the CPS relation in the variable environment extended with the function name and the function argument.

**CPS conversion of top-level program.** Lastly, we define a top-level CPS transformation that instantiates the fresh set with the set of variables that are greater than all the variables that are already used in the variable environment and the current continuation. That is:

$$e; \bar{x}; k \triangleright \bar{e}' \overset{\text{def}}{=} \exists S', \{y \mid \forall z \in \{k, \bar{x}\}, z < y\}; e; \bar{x}; k \triangleright \bar{e}'; S'$$

### 6 THE LOGICAL RELATION

In order to prove correctness for our transformation, we use an untyped, step-indexed logical relation. [Ahmed 2006; Appel and McAllester 2001]. Although logical relations are most frequently indexed by types, untyped logical relations are not uncommon [Acar et al. 2008; Owens et al. 2017; Paraskevopoulou and Appel 2019].
The definition of the logical relation (fig. 7) consists of an expression relation, a value relation, a result relation, and an environment relation. The expression relation \( E^k(\sigma_1, e_1) \sim (\sigma_2, e_2) \{ Q_L : Q_G \} \) relates two target configurations (pairs of an environment and an expression) at a given step index. As usual, the relation states that if the first configuration evaluates to some result in some number of steps less than the given step index, then so does the second expression and furthermore the two results are related for the remaining number of steps. As we explained in section 3, the logical relation is indexed by two binary relations on fuel values. The local invariant, \( Q_L \), is imposed on the fuel consumption of the current configurations. The global invariant, \( Q_G \), holds for future executions of the results and holds for whole function bodies.

Similar relations indexed by local and global invariants have been used in other parts of CertiCoq in order to prove that transformations preserve terminating and diverging executions and to derive upper bounds on the resource consumption of programs [Paraskevopoulou and Appel 2019; Paraskevopoulou et al. 2021]. The logical relation we present here is used extensively to prove correct transformations in the \( \text{\textcdot}_\text{ANF} \) (\( \text{\textcdot}_\text{CPS} \) is a subset of \( \text{\textcdot}_\text{ANF} \)) middle-end pipeline of CertiCoq. However, it is used in a more traditional way where source and target programs are expressed in the same language and are proved related with the logical relation.

The value relation, written \( V^k(v_1, v_2) \{ Q \} \) relates two values at a given step-index \( k \) and global postcondition \( Q \). Two constructors are related if their arguments are pairwise related values. Two closures are related if for any two lists of parameters related at some strictly smaller step index \( i < k \), 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. Observe that when the expression relation is invoked, the local and global invariants are instantiated with the same (global) invariant. The result relation lifts the value relation to results. Two results are related if they are either out-of-time exceptions or related values. The environment relation \( S \triangleq f \ M^k(\sigma_1, \sigma_2) \{ Q \} \) relates two environments under a set of variables \( S \), a variable renaming \( f \), the step-index \( k \) and the global postcondition \( Q \). It asserts that for any variable \( x \) in the set \( S \), if \( x \) is defined in the domain of \( \sigma_1 \) then \( f(x) \) defined in \( \sigma_2 \) and furthermore their values are related with the value relation.

**Invariant notations.** We reuse the invariant notations of section 3. We write \( m = \) for the invariant \( \text{fun} \ c_1, c_2 \Rightarrow c_1 + m = c_2 \) and simply \( = \) for the invariant \( 0 = \). We write \( m \leq \) (resp. \( m \geq \)) for the invariant \( \text{fun} \ c_1, c_2 \Rightarrow c_1 + m \leq c_2 \) (resp. \( \text{fun} \ c_1, c_2 \Rightarrow c_1 + m \geq c_2 \)). We write \( Q_1 \cap Q_2 \) to denote the conjunction of \( Q_1 \) and \( Q_2 \).

### 6.1 Properties of the Logical Relation

**Compatibility properties.** As usual, the logical relation is compatible with the language constructs. The compatibility lemmas assert that language constructs preserve the logical relation. However, in this setting we need to make some assumptions about the local and global invariants for the compatibility lemmas to hold. A local invariant \( Q_L \) and a global invariant \( Q_G \) that satisfy the assumptions are needed to prove the compatibility lemmas satisfy \( \text{Compat} Q_G Q_L \). The \( \text{Compat} Q_G Q_L \) predicate asserts that:

1. \( Q_L (0, 0) \) i.e., the local invariant holds for zero fuel. This is useful to relate two executions that throw an out of time exception.
2. \( Q_L (1, 1) \) i.e., the local invariant holds for unit fuel. This is useful to establish a compatibility lemma for return.
3. If \( Q_L (c_1, c_2) \) then \( Q_L (c_1 + a, c_2 + a) \), i.e. the local invariant is preserved by adding a constant.

This is useful to establish compatibility rules for the inductive cases of the evaluation relation.
Value relation

\[ V^k(C_1(\overline{v_1}), C_2(\overline{v_2})) \{ Q \} \overset{\text{def}}{=} C_1 = C_2 \wedge V^k(\overline{v_1}, \overline{v_2}) \{ Q \} \]

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

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

\[ V^k(v_1, v_2) \{ Q \} \overset{\text{def}}{=} \text{False} \quad \text{Otherwise.} \]

Result relation

\[ R^k(\text{OOT, OOT}) \{ Q \} \overset{\text{def}}{=} \text{True} \]
\[ R^k(\text{Res}(v_1), \text{Res}(v_2)) \{ Q \} \overset{\text{def}}{=} V^k(v_1, v_2) \{ Q \} \]
\[ R^k(r_1, r_2) \{ Q \} \overset{\text{def}}{=} \text{False} \quad \text{Otherwise.} \]

Expression relation

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

Environment relation

\[ S \vdash f \ C^k(\sigma_1, \sigma_2) \{ Q \} \overset{\text{def}}{=} \forall (x \in S) v_1, \sigma_1(x) = v_1 \Rightarrow \exists v_2, \sigma_2(f(x)) = v_2 \wedge V^k(v_1, v_2) \{ Q \} \]

Fig. 7. The logical relation.

Local invariant monotonicity. The logical relation is monotonic on the local invariant.

**Lemma 6.1 (Local invariant monotonicity).** Assume that \( Q_L \subseteq Q_{L'} \).
If \( E^k(\sigma_1, e_1) (\sigma_2, e_2) \{ Q_L; Q_G \} \), then \( E^k(\sigma_1, e_1) (\sigma_2, e_2) \{ Q_{L'}; Q_G \} \).

The logical relation is not monotonic in the global invariant that appears in both negative and positive positions in the definition of the value relation.

Reflectivity and transitivity. The logical relation is reflexive and transitive. As we explained, both of these properties are crucial for the CPS proof.

**Lemma 6.2 (Reflexivity).** Assume that \textit{Compat} \( Q_G \) \( Q_L \).
For any two related environments \( \text{fv}(e) + C^k(\sigma_1, \sigma_2) \{ Q_G \} \), we have that \( E^k(\sigma_1, e) (\sigma_2, e) \{ Q_L; Q_G \} \).

This lemma is proved using the compatibility properties of the relation.

**Lemma 6.3 (Transitivity).** Assume that \textit{Compat} \( Q_G \) \( Q_G \) and \( Q_G \circ Q_G \supseteq Q_G \).
If \( E^k(\sigma_1, e_1) (\sigma_2, e_2) \{ Q_1; Q_G \} \) and \( E^k(\sigma_1, e_1) (\sigma_2, e_2) \{ Q_2; Q_G \} \) then \( E^k(\sigma_1, e_1) (\sigma_2, e_2) \{ Q_1 \circ Q_2; Q_G \} \)

The local invariant in the conclusion of the above lemma is (unsurprisingly) the composition of the two local invariants as the premises. The global invariant is the same as in the premises. The lemma requires that the global invariant satisfies the compatibility properties (as it uses reflexivity to show that the value relation is transitive), and that composition of \( Q_G \) with itself implies \( Q_G \) (which is needed to apply the induction hypothesis).
**Small-step lemmas.** A key observation for our reasoning, is that we can use this logical relation to transition between intermediate states of computations, something that is not possible with bare big-step semantics. In particular, we can state lemmas that relate two configurations before and after evaluating a language construct. For application we have:

**Lemma 6.4 (Reduce-App).** If $\sigma(f) = \text{Clos}(\sigma, \text{fun } g \overline{y} = e_g)$ and $\sigma(\overline{x}) = \overline{v}$ then

$$E(\sigma \overline{y} \mapsto \overline{v}, g \mapsto \text{Clos}(\sigma, \text{fun } g \overline{y} =)), e_g) (\sigma, f \overline{x}) \{^1;=;=\}$$

This lemma relates a configuration before and after evaluating a function application. The right-hand side configuration is the function application whereas the left-hand side is the function body in the closure environment extended with the appropriate bindings. The other direction could be proved too, but for the sake of this proof this is enough.

This lemma also demonstrates why we need to distinguish between local and global invariants. The two configurations that we relate in the above lemma compute exactly the same value. Therefore the fuel consumption of whatever closure values are in the result is exactly the same. However, the fuel consumption is not the same for the two related configurations. If the left-hand side computes a value in $c$ steps then the right-hand side computes the same value in $c + 1$ steps since it has to evaluate the function application in addition to all the evaluation of the function body.

Similarly, we can define such a lemma for each language construct. For our proof we only need rules for function definition and constructor in addition to the one for function application shown above. These are stated below.

**Lemma 6.5 (Reduce-Constr).** If $\sigma(\overline{y}) = \overline{v}$ then

$$E(\sigma[x \mapsto C(\overline{v})], e) (\sigma, \text{let } x = C(\overline{y}) \text{ in } e) \{^1;=;=\}$$

**Lemma 6.6 (Reduce-Fun).** $E(\sigma[f \mapsto \text{Clos}(\sigma, \text{fun } f \overline{x} = e_f)], e) (\sigma, \text{fun } f \overline{x} = e_f \text{ in } e) \{^1;=;=\}$

We now have all the machinery that we need to state and prove correctness for the CPS transformation. This is the subject of the next section.

7 CORRECTNESS THEOREM AND PROOF

In this section we present the main results for the CPS translation. We show the key facts that we need in order to establish the simulation proof of the CPS-transformation. Then, we show how we can use the main correctness theorem to derive the top level corollary for both whole-program and separate compilation.

**Freshness assumptions.** In the theorems that follow, we will need to assume some freshness requirements for the parameters of the CPS relation. Given a CPS translation judgment $S; e; x; k \triangleright e'; S'$, we will need that the fresh set $S$ does not contain any variables that are already used in the variable environment $\overline{x}$ or the continuation $k$, and that that elements of $\overline{x}$ are pairwise disjoint and also disjoint from $k$. We define the predicate $\text{fresh}(S, \overline{x}) \overset{\text{def}}{=} S \cap \{\overline{x}\} = \emptyset \land \text{NoDup}(\overline{x})$ over a set of variables $S$ and a list of variables $\overline{x}$. It asserts that the list of variables is pairwise disjoint (using the predicate NoDup) and also disjoint from the fresh set of variables.

**Well-scopedness.** For the main theorem of the CPS transformation we also need to assume that the source program is well-scoped. We define $\text{well\_scoped}(e, \overline{x})$ to mean that all the free (De Bruijn) variables of the expression $e$ are defined in the environment $\overline{x}$. Note that only the length of the environment is relevant in this definition. Therefore, $\overline{x}$ can be either a list of values or list of variables. We also define a notion of well-scoped values and environments (using the same overloaded meta-function name). $\text{well\_scoped}(v)$ holds if a value is a constructor whose arguments are well-scoped values, or a closure whose function part is well-scoped in its environment. We define $\text{well\_scoped}(\overline{v})$ to be the pointwise lifting of value well-scopedness to environments.
7.1 \( \alpha \)-Equivalence

As in the proof of section 2, we will need to state and prove that CPS-translating a source term will produce logically related target terms for any choice of variable names that are sufficiently fresh. This is captured by the following lemma.

**Lemma 7.1 (Fresh name invariance).** Assume that

1. \( S_1; e; k_1 \triangleright e_1'; S_1' \) and fresh \( (S_1, k_1 : \overline{x}) \)
2. \( S_2; e; k_2 \triangleright e_2'; S_2' \) and fresh \( (S_2, k_2 : \overline{x}) \)
3. \( \{ k_1, \overline{x} \} + id[k_1\mapsto k_2, \overline{x}\mapsto \overline{y}] \quad \mathcal{C}(\sigma_1, \sigma_2) \{ Q_G \}, \) and
4. \( \text{Compat} \{ Q_G \} \{ Q_L \} \).

Then, we have that \( \mathcal{E}(\sigma_1, e_1) \quad (\sigma_2, e_2) \{ Q_L; Q_G \} \).

The first two assumptions state that the term \( e \) is CPS-translated with some fresh set, variable environment, and continuation variable that are fresh. The fourth assumption states that the two target environments \( \sigma_1 \) and \( \sigma_2 \) map the continuation variable and the variables in the two variable environments to logically related values. Note that at the top-level the variable environments are empty and the continuation variables are mapped to the identity function so this is trivially true. The conclusion states that the two configurations obtained by CPS-translated terms and the two logically related environments are logically related. The proof of this lemma is straightforward but somewhat tedious. It is proved by induction on the step-index and case analysis on terms. Each case follows directly by the application of the compatibility lemmas. Hence, we need the third assumption. Observe that this theorem does not put any additional restrictions on the local and global invariants other than they satisfy the compatibility conditions.

7.2 Main Theorem

For the main theorem of CPS, we need to distinguish two cases. First, we need a theorem that asserts what happens when the source terminates to a value. Second, we need a theorem that asserts what happens when the source evaluation terminates with an out-of-time exception. The second theorem is crucial in order to establish that diverging executions are preserved.

**Termination.** The correctness theorem for terminating executions states the following:

**Lemma 7.2 (Correctness, termination).** Assume that

1. \( S_1; e_1; k_1 \triangleright e_1'; S_1' \) (\( e_1' \) is the CPS-translation of \( e \))
2. \( \text{fresh}(S, k_1 : \overline{x}) \) (the fresh set \( S \), variable environment \( \overline{x} \) and continuation \( k_1 \) are fresh)
3. \( \text{wellScoped}(e, \overline{x}) \) and \( \text{wellScoped}(\overline{y}) \) (the source program and environment are well-scoped)
4. \( \overline{v}; e \upharpoonright _{\text{src}}(\overline{c}) \) \( \text{Res}(v) \) (the source program evaluates in the environment \( \overline{v} \) to the value \( v \)).

Then for all target environments \( \sigma \) such that \( \overline{x}, \overline{y} \triangleright \sigma \), target values \( v' \) such that \( v \triangleright v' \), and continuations \( v_k \), we have \( \mathcal{E}(k, y, [k \mapsto v_k, y \mapsto v']) \quad (e', \sigma[k \mapsto v_k]) \{ f \leq \wedge \leq \geq ; = \}

The result of the lemma states that for any target environment \( \sigma \) that is CPS-related to the the variable environment \( \overline{x} \) and value environment \( \overline{y} \), for any target value \( v' \) that is CPS-related to the source result \( v \), and for any continuation value \( v_k \), the term \( k \) \( y \) in an environment that maps \( y \) to \( v' \) and \( k \) to \( v_k \) is logically related to the CPS-translated term \( e' \) in the environment \( \sigma \) (that contains the free variables of the term) extended to map the continuation variable \( k \) to \( v_k \). At the top-level we instantiate \( v_k \) with the identity continuation and therefore the term \( k \) \( y \) evaluates to the value \( v' \). This lets us obtain our top-level result.

The invariant \( f \leq \wedge \leq \geq ; = \) that we impose on the two related terms states that the fuel consumption \( f_1 \) of the term \( k \) \( y \) and the fuel consumption \( f_2 \) of the term \( e' \) satisfy (1) \( f_1 + c \geq f_2 \), and (2) \( f_1 + f \leq f_2 \), where \( f \) is the fuel consumption of the source program and \( c \) the cost of the source program. At
the top-level, \( k \) will be bound to the identity continuation so the fuel consumption \( f_1 \) of \( k \) \( y \) will be equal to two units of time (one for function application and one for function return). Therefore we can derive that \( f_2 \leq c + 2 \), which we use to show an upper bound on the execution cost of the CPS program. We also derive that \( f \leq f_2 \) which we use to show divergence preservation.

The proof of this theorem proceeds by induction on the evaluation derivation. In each case of the proof we transition, using transitivity, from the right-hand side term to some other term until we reach the left-hand side term. Each transition follows using one of the: induction hypotheses, small-step lemma (of section 6.1), application compatibility, or fresh name invariance (lemma 7.1).

We found that the resource invariants were quite easy to establish: in each case of the proof we first use the local invariant monotonicity property (lemma 6.1). This generates an uninstantiated local invariant and a proof obligation that it is stronger than the one we are trying to establish. At the end of each case, the uninstantiated invariant is instantiated as the composition of all intermediate invariants for each step we transitively composed. The proof obligation that the composition implies the invariant we want to establish is solved automatically using Coq’s lia tactic for linear integer arithmetic.

**Divergence.** The theorem that we use to establish that divergence is preserved states that whenever the source execution times out for some amount of fuel, then so does the target term for some target fuel that is lower bounded by the source fuel. Stated formally:

**Lemma 7.3 (Correctness, out-of-time). Assume that**

1. \( S; e; \bar{x}; k \triangleright e'; S' \)
2. fresh\((S, k :: \bar{x})\)
3. wellScoped\((e, \bar{v})\) and wellScoped\((\bar{v})\)
4. \( \bar{v}; e \triangleright \text{src} \) OOT.

**Then then for all target environments \( \sigma \) such that \( \bar{x}; \bar{v} \triangleright \sigma \) and continuations \( \nu_k \), we have that there exists \( f' \) such that \( f \leq f' \) and \( (\sigma[k \mapsto \nu_k], e') \triangleright \text{trg} \) OOT.**

This lemma is proved using lemma 7.2. In our mechanized development we show these two facts simultaneously so we can factor out some common parts of the two proofs. Using the above lemma and lemma 4.2, we can easily derive that if \( \bar{v}; e \triangleright \text{src} \) then \( \rho[k \mapsto \nu_k]; e' \triangleright \text{trg} \).

### 7.3 Top-Level Correctness

In order to compose the correctness specification of CPS with the rest of the proofs of our compiler, we need to derive a correctness theorem for the top-level CPS transformation.

For whole program compilation, we want to say that whenever the source terminates to a value then so does the target and furthermore, that the target value refines the source value and the fuel consumption of the target program is upper bounded by the cost of the source program (plus a constant). When the source program diverges, then the target must diverge too.

The value refinement states that constructed values are related if they have the same outermost constructor and pairwise related arguments. A target (CPS) closure is trivially refined by any source closure. Therefore, when we run a whole program we can only observe the first-order parts of a result. A higher-order result can observed only by another program that applies the function to some argument. This is covered by our top-level theorem about separate compilation. The value refinement relation is shown below.

\[
C_j(v_1, \ldots, v_m) \approx C_j(v'_1, \ldots, v'_n) \overset{\text{def}}{=} C_j = C_j \land n = m \land \forall i, \: v_i \approx v'_i
\]

\[
\text{Clos}(\bar{v}, \text{fix}(e)) \approx \text{Clos}(\sigma, \text{fun} \ f \ k \ x = e') \overset{\text{def}}{=} \text{True}
\]
Using the value refinement relation, we can state the whole program refinement between the source and target languages as we described it above.

\[
e_{\text{src}} \triangleright_{B} e_{\text{trg}} \overset{\text{def}}{=} \left( \forall c_{\text{src}} \in \text{src}, e_{\text{src}} \parallel\parallel_{c_{\text{src}}} \text{Res}(\nu_{\text{src}}) \Rightarrow \right.
\]
\[
\exists f_{\text{trg}} \nu_{\text{trg}}, e_{\text{trg}} \parallel_{f_{\text{trg}}} \text{Res}(\nu_{\text{trg}}) \land \nu_{\text{src}} \approx \nu_{\text{trg}} \land f_{\text{trg}} \leq c_{\text{src}} + 3 \land
\]
\[
\left( e_{\text{src}} \triangleright \Rightarrow e_{\text{trg}} \triangleright \right)
\]

The constant 3 that is added to the cost of the source program is needed for the definition and application of the top-level continuation.

7.3.1 Whole Programs. From lemma 7.2 and lemma 7.3 we can derive the following top-level correctness theorem.

**Theorem 7.4 (Whole-program correctness).**

If \(\text{well-scoped}(e, [])\) then there exists \(e'\) such that \(e; [] \triangleright e'\) and then \(e \triangleright_{B} \text{fun} k x = \text{ret} x \in e'\).

That is, a well-scoped source program can always be CPS-converted (with the top-level CPS transformation) to a target program—because we have proved that the CPS relation is total on well-scoped terms— and the program refines the source program in a context where the continuation variable \(k\) is defined to be the identity function. To derive the program refinement for terminating programs, we instantiate \(\nu_{e}\) in lemma 7.2 to be the closure \(\text{Clos}(\cdot, \text{fun} k x = x)\). Then, the configuration \((k y, [k \mapsto \text{Clos}(\cdot, \text{fun} k x = x), y \mapsto v'])\) terminates to \(\text{Res}(v')\) for which we know that \(v \triangleright v'\), where \(v\) is the result of source evaluation. Then the configuration \(([k \mapsto \text{Clos}(\cdot, \text{fun} k x = \text{ret} x)], e')\) terminates to a value \(v''\) such that \(V(v', v'') \{=\}\). By using the fact that \(v \triangleright v' \Rightarrow V(v', v'') \{=\} \Rightarrow v \approx v''\), we obtain the value refinement.

7.3.2 Separate Compilation. Before stating the separate compilation theorem, we must define a linking operation in the source and target languages. In the source language, we can link some program \(e_{\text{client}}\) with a free De Bruijn variable with the program \(e_{\text{lib}}\) that computes the value of the free variable by let-binding \(e_{\text{lib}}\) in \(e_{\text{client}}\). That is:

\[
[e_{\text{lib}}]e_{\text{client}} \overset{\text{def}}{=} \text{let } e_{\text{lib}} \text{ in } e_{\text{client}}
\]

In the target, we define the linking some program \(e_{\text{client}}\) with a free variable \(x\) with the program \(e_{\text{lib}}\) that computes the value of the free variable \(x\) as

\[
[x \mapsto e_{\text{lib}}]e_{\text{client}} \overset{\text{def}}{=} \text{fun} k_{1} x = \text{fun} k_{2} y = \text{ret} y \text{ in } e_{\text{client}} \text{ in } e_{\text{lib}}
\]

where \(k_{1}\) (resp. \(k_{2}\)) is the continuation variable that was used in translating \(e_{1}\) (resp. \(e_{2}\)) and \(y\) is a fresh variable. Intuitively, the continuation of the library program will run the client program with \(x\) bound to the result of the library program. The continuation of the client program is the identity continuation.

Using these operations, we can show that the result of linking two separately CPS-translated programs in the target, refines the result of linking the two programs in the source.

**Theorem 7.5 (Separate-compilation correctness).**

If \(\text{well-scoped}(e_{\text{lib}}, [])\) and \(\text{well-scoped}(e_{\text{client}}, [x])\), and \(x, k_{1}\), and \(k_{2}\) are pairwise disjoint, then there exist CPS translations \(e'_{\text{lib}}\) and \(e'_{\text{client}}\), such that \(e_{\text{lib}}; [] \triangleright e'_{\text{lib}}, e_{\text{client}}; [x] \triangleright e'_{\text{client}}\), and \([e_{\text{lib}}]e_{\text{client}} \triangleright_{B} [x \mapsto e'_{\text{lib}}]e'_{\text{client}}\).
7.4 Discussion
The proof technique we presented deals elegantly with two of the most complex aspects of the correctness of CPS transformation: the problem of $\alpha$–conversion and the failure of certain translations to preserve reductions in the source language. Other solutions have been suggested too, but they focus on using different binder representations—De Bruijn and (parametric) higher-order abstract syntax— to deal with $\alpha$–conversion, and ad-hoc simulation relations that involve more complicated notions of reductions, colon translations and custom equality notions. We consider our technique more systematic as it does not depend on the specific translation that is being proved correct and uses a straightforward name binder representation that is the most in compilers.

**Logical relations vs. $\alpha$–equivalence.** We find that using a logical relation has many benefits compared to explicitly accounting for $\alpha$–equivalence. With a logical relation, we can establish a much simpler simulation diagram leading to a simpler proof and a more compositional style of reasoning. In particular, we hide the existential quantifier inside the definition of the logical relation and we use the theory of the logical relation to compose different proof steps without needing to extract an the existential witness at each step (recall the statement and proof of the CPS transformation in section 2.1 when $\alpha$-equivalence is explicitly used). Although we have to define a logical relation, our logical relation is reusable in many contexts (the same logical relation has been already used to prove correctness for seven other passes of the CertiCoq compiler [Paraskevopoulou et al. 2021]).

**Direct-style transformations.** Even though we only use this technique to prove correctness of a CPS transformation, we conjecture that the same technique can be used to prove correct other cross-language transformations with similar benefits. In the case of a direct-style transformation the conclusion of lemma 7.2 would have to be modified to $E(y, [y \mapsto v']) \ (e', \sigma) \ \{Q; Q_G\}$, where $v'$ is a target value related to the result of the source evaluation, $e'$ and $\sigma$ the target term and environment, and $Q, Q_G$ an appropriate choice of invariants.

8 RELATED WORK
Formalizing the correctness of a CPS transformation (in a proof assistant or otherwise) is not an easy task and it has been extensively studied. As we discussed, the main source of complication is that many CPS translations do not preserve reductions in the source language.

The first semantics preservation proof for CPS is given by Plotkin [1975]. He shows, with a simulation argument, that the translation produces correct outputs and that evaluation yields the same result under call-by-name and call-by-value reduction. The transformation produces administrative redexes that complicates reasoning. To establish the results he defines a colon-translation that reduces some administrative redexes. To address the difficulties introduced by administrative reductions, Danvy and Nielsen [2003] provide a one-pass, first-order, compositional CPS transformation (previous one-pass CPS transformations were either higher-order or non-compositional and have complicated correctness proofs). Their transformation satisfies a simple simulation diagram.

Several authors have attempted mechanizing the correctness of CPS transformations in different settings.

**Named representations.** Minamide and Okuma [2003] formalize the correctness of three CPS transformations (those of Plotkin [1975], Danvy and Nielsen [2003], and Danvy and Filinski [1992]) in Isabelle/HOL. To avoid complexities related to De Bruijn indices they use named variables. However, they must formalize and take explicit account of $\alpha$–equivalence in their proofs.

**De Bruijn representations.** Dargaye and Leroy [2007] mechanize in Coq the correctness of Plotkin’s original call-by-value transformation and Danvy and Nielsen’s one-pass transformation.
They use De Bruijn indices to represent binders, but they resort to a non-standard representation with two distinct name spaces in order to efficiently implement the transformation. They use big-step semantics (to avoid difficulties related to administrative redexes) and therefore they do not show that divergence is preserved. Because the one-pass transformation does not commute with substitution, they do not give a direct proof of correctness. They rather show that the one-pass transformation reduces to the naive transformation using call-by-value parallel reduction. Pottier [2017] provides a one-pass, compositional formulation of Danvy and Filinski’s CPS transformation and mechanize its correctness in Coq. He uses De Bruijn indices to represent binder and also deals with difficulties in implementing and proving correct an efficient transformation. He first shows that an inefficient first-order formulation of the transformation is correct. He notes that in the presence of a let-construct the standard simulation diagram does not hold anymore, and he establishes a more complicated diagram that employs parallel call-by-value reduction. In order to prove that diverging behaviors are preserved he needs to mechanize that parallel call-by-value reduction commutes with call-by-value reduction. He then provides a efficient \( O(n \log(n)) \) higher-order implementation of the transformation (that as he notes is hard to discover and not clear how it works) that he proves equivalent to the inefficient transformation. Chlipala [2007] mechanizes the correctness of a CPS transformation in Coq using De Bruijn indices, in the context of a verified type-preserving compiler for simply-typed lambda calculus. To prove the transformation correct, he defines a type-directed interpretation of terms in the meta-language (Gallina) and he uses a type-directed logical relation to establish that the source and target terms have the same denotation. This proof relies on the fact that the language is typed and strongly normalizing.

(Parametric) higher-order abstract syntax. Tian [2006] uses higher-order abstract syntax to mechanize the correctness of a one-pass CPS transformation in Twelf avoiding the issues related to \( \alpha \)–conversion. He is able to establish a simple simulation diagram but he uses big-step semantics in the source language. Therefore, his result applies only to terminating source and target programs. Chlipala [2008] introduces parametric higher-order abstract syntax (PHOAS) and uses it to prove correct CPS transformations for simple-typed lambda calculus and system F. The proof technique is the same as in his previous paper [2007] and therefore applies only to strongly normalizing languages. Yamada and Asai [2018] uses PHOAS to mechanize the correctness of a CPS transformation for let-polymorphic lambda calculus in Agda. Because the translation does not preserve reductions in the source language, they define a custom notion of equality on target terms, which is translation dependent and entails \( \beta \)–equality, and use to build a simulation diagram. This simulation diagram cannot be used to show that diverging behaviors are preserved.

Resource preservation. Minamide [1999] shows that a CPS transformation preserves the observable behaviors of the source program and also the space (but not the time) required for its execution (within a constant factor). The simulation proof, which is not mechanized, uses big-step semantics in the source and it does not generalize to non-terminating executions of source programs.

Other properties of CPS have been formalized too. A number of authors have investigated properties like equivalence preservation [Ahmed and Blume 2011; Sabry and Felleisen 1992] and type preservation [Bowman et al. 2017; Harper and Lillibridge 1993; Morrisett et al. 1999] but these are not it the scope of this paper.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1521602 and Grant No. 2030859 to the Computing Research Association for the CIFellows Project. We wish to thank Andrew W. Appel, John M. Li, Daniel Patterson, Andrew Wagner, and the anonymous reviewers for their feedback on various versions 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 Proceedings of the 15th European Conference on Programming Languages and Systems (Vienna, Austria) (ESOP’06). Springer-Verlag, Berlin, Heidelberg, 69âĂŞ83. https://doi.org/10.1007/11693024_6

Amal Ahmed and Matthias Blume. 2011. An Equivalence-Preserving CPS Translation via Multi-Language Semantics. In Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (Tokyo, Japan) (ICFP ’11). Association for Computing Machinery, New York, NY, USA, 431–444. https://doi.org/10.1145/2034773.2034830

Abhishek Anand, Andrew W. Appel, John Gregory Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélangé, 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. 1997. 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 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

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In Theorem Proving in Higher Order Logics, Joe Hurd and Tom Melham (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–65. https://doi.org/10.1007/11541868_4

William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2017. Type-Preserving CPS Translation of Î£ and Î± Types is Not Not Possible. Proc. ACM Program. Lang. 2, POPL, Article 22 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158110

Adam Chlipala. 2007. A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (San Diego, California, USA) (PLDI ’07). Association for Computing Machinery, New York, NY, USA, 54–65. https://doi.org/10.1145/1250734.1250742

Adam Chlipala. 2008. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (Victoria, BC, Canada) (ICFP ’08). Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/1411204.1411226

Oliver Danvy and Andrzej Filinski. 1992. Representing Control: a Study of the CPS Transformation. Mathematical Structures in Computer Science 2, 4 (1992), 361–391. https://doi.org/10.1017/S0960129500001535

Oliver Danvy and Lasse R. Nielsen. 2003. A first-order one-pass CPS transformation. Theoretical Computer Science 308, 1 (2003), 239–257. https://doi.org/10.1016/S0304-3975(02)00738-8

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’07). Springer-Verlag, Berlin, Heidelberg, 211–225.

Timothy G. Griffin. 1989. A Formulae-as-Type Notion of Control. In Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (San Francisco, California, USA) (POPL ’89). Association for Computing Machinery, New York, NY, USA, 47–58. https://doi.org/10.1145/96709.96714

Robert Harper and Mark Lillibridge. 1993. Explicit Polymorphism and CPS Conversion. In Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Charleston, South Carolina, USA) (POPL ’93). Association for Computing Machinery, New York, NY, USA, 206âĂŞ219. https://doi.org/10.1145/158511.158630

Andrew Kennedy. 2007. Compiling with Continuations, Continued. In Proceedings of the 20th 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

Xavier Leroy. 2009. A Formally Verified Compiler Back-End. J. Autom. Reason. 43, 4 (Dec. 2009), 363Â­Â–446. https://doi.org/10.1007/s10817-009-9155-4

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

Yasuhiko Minamide and Koji Okuma. 2003. Verifying CPS Transformations in Isabelle/HOL. In Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding (Uppsala, Sweden) (MERLIN ’03). Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/976571.976576

Proc. ACM Program. Lang., Vol. 5, No. OOPSLA, Article 114. Publication date: October 2021.
Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.* 21, 3 (May 1999), 527–568.  
https://doi.org/10.1145/319301.319345

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.  
https://doi.org/10.1007/978-3-662-49498-1_23

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/10.1145/3102626

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. [OOPSLA 2021 Artifact] Compiling With Continuations, Correctly.  
https://doi.org/10.5281/zenodo.5504155

Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. 2021. Compositional Optimizations for CertiCoq. *Proc. ACM Program. Lang.* 3, ICFP, Article 86 (July 2021), 30 pages.  
https://doi.org/10.1145/3341687

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

François Pottier. 2017. Revisiting the CPS Transformation and its Implementation. *Unpublished* (2017).

Amr Sabry and Matthias Felleisen. 1992. Reasoning about Programs in Continuation-Passing Style. *SIKPLAN Lisp Pointers* V, 1 (Jan. 1992), 288–298.  
https://doi.org/10.1145/141478.141563

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

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

Ye Henry Tian. 2006. Mechanically Verifying Correctness of CPS Compilation. In *Proceedings of the Twelfth Computing: The Australasian Theory Symposium - Volume 51* (Hobart, Australia) (CATS ’06). Australian Computer Society, Inc., AUS, 41–51.

Urara Yamada and Kenichi Asai. 2018. Certifying CPS Transformation of Let-Polymorphic Calculus Using PHOAS. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 375–393.  
https://doi.org/10.1007/978-3-030-02768-1_20

Proc. ACM Program. Lang., Vol. 5, No. OOPSLA, Article 114. Publication date: October 2021.