mathematics

Article

Coinductive Natural Semantics for Compiler Verification in Coq [†]

Angel Zúñiga [1,*] and Gemma Bel-Enguix [2]

1 Posgrado en Ciencia e Ingeniería de la Computación, Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas, Universidad Nacional Autónoma de México, Mexico City 04510, Mexico
2 Instituto de Ingeniería, Universidad Nacional Autónoma de México, Mexico City 04510, Mexico; gbele@iingen.unam.mx
* Correspondence: zuniga@ciencias.unam.mx

† This paper is an extension of: Zúñiga, A.; Bel-Enguix, G. A Correct Compiler from Mini-ML to a Big-Step Machine Verified Using Natural Semantics in Coq. In Proceedings of the XVIII Jornadas de PROgramación y LEnguajes (PROLE 2018), Seville, Spain, 17–19 September 2018.

Received: 4 July 2020; Accepted: 9 September 2020; Published: 12 September 2020

Abstract: (Coinductive) natural semantics is presented as a unifying framework for the verification of total correctness of compilers in Coq (with the feature that a verified compiler can be obtained). In this way, we have a simple, easy, and intuitive framework; to carry out the verification of a compiler, using a proof assistant in which both cases are considered: terminating and non-terminating computations (total correctness).

Keywords: natural semantics; big-step semantics; coinduction; compiler verification; total correctness; Mini-ML; SECD machine; Coq proof assistant

MSC: 68Q55; 68N20; 68Q60; 68N15; 68N18; 03B35

1. Introduction

This paper tackles the problem of compiler verification in proof assistants. At present, a number of long-term projects deals with several aspects of this issue, for instance, CompCert C [1–5], CertiCoq [6], and IRIS [7]. In this work, we address specifically the verification of total correctness of compilers of functional languages in Coq. Here, we refer to total correctness in the sense of Leroy [8] and Gregoire and Leroy [9], that is total correctness means: correctness of terminating and non-terminating computations.

In the literature, ad-hoc verifications are traditionally used; meaning, verifications that employ more than one distinct formalism.

This situation calls for a solution that abstracts away everything needed in a single unifying framework that simplifies (and perhaps, it could even make possible to automate) this task. By unifying framework we mean a single formalism able to define each of the components of a compiler, namely: source language semantics, intermediate language semantics, abstract machine semantics, and translations. In fact, in this work we offer (coinductive) natural semantics as a simple, easy and intuitive unifying framework to carry out (total correctness) compiler verification in Coq. Whenever we use 'framework', and we are referring to the (coinductive) natural semantics unifying framework as presented in this work (and only to it), we mean 'unifying framework'. In this manner, we remark that only one formalism, (coinductive) natural semantics, is sufficient to conduct this task as opposed to usual verifications in the literature where more than one distinct formalism are needed in order to accomplish the same goal (see Section 1.1 for a discussion of the related work). In our preliminary work [10] we only address correctness for termination. In this extended and improved version, as the main novelty, we also tackle correctness for non-termination (which is the most challenging one).

Natural semantics, as introduced by Kahn [11], was initially a unifying framework. Indeed, since its inception [11], natural semantics has been able to express: source language semantics, intermediate languages semantics, abstract machine semantics and translations. Despeyroux [12] also showed how natural semantics allows proof of the correctness of the translations. Based on the work of Kahn and Despeyroux, Boutin [13] formalized a compiler from Mini-ML to the CAM in Coq (although it is not possible to obtain a verified compiler from his formalization).

It is folklore that natural semantics, as originally introduced by Kahn [11], is unqualified to express non-terminating computations (the articles of Kahn, Despeyroux, and Boutin, only address the case of terminating computations). More precisely, Despeyroux made a preliminary attempt to cover infinite computations but it is based on denotational semantics instead of natural semantics per se. However, since the introduction of coinductive natural semantics by Leroy [14] and Leroy and Grall [15] it is a well-known practice to use a coinductively defined natural semantics to deal with non-terminating program behavior. Leroy [14] and Leroy and Grall [15] advance coinductive natural semantics as a formalism to express non-terminating computations; however, they do not use coinductive natural semantics as a unifying framework in the original sense of Kahn, but they use it only to specify the source language (a high-level language). Specifically, they make an ad-hoc formalization in which in addition, they use small-step semantics in (what was later called) the Modern SECD (MSECD) machine [8] and a function to define the compilation.

The main idea of this work is to extend natural semantics so that it becomes a simple, easy, and intuitive unifying framework for verifying total correctness of compilers in Coq (with the ability that you can get a verified compiler that can be used in real life).

The strategy for carrying out this extension is as follows: until now, taking as reference the work of Boutin, natural semantics can be used for formalizing the correctness of a compiler in Coq (only for the termination case and a verified compiler cannot be obtained). The first thing we will do is to show how, from a natural semantics specification, one can obtain, either an interpreter (Section 2.1) or a compiler (Section 2.3.1) sound and complete regarding the natural semantics specification. Then, for natural semantics to be able to express non-terminating computations, we will use Leroy's coinductive natural semantics. However, this semantics has only been used in the source language, in addition, we will show how to use coinductive natural semantics to specify non-terminating computations in an abstract machine (Section 3.3). We will also show that an interpreter, which was previously obtained from natural semantics, is sound and complete regarding its corresponding coinductive natural semantics specification (Section 3.1), that is, for the non-termination criteria (and not only for the termination case). In this way, we are ready to formulate and prove the correctness (semantic preservation) of a compiler, both for the termination case (Section 2.3.2), and, for the non-termination case (Section 3.3.2), completely in terms of (coinductive) natural semantics.

To illustrate the use of (coinductive) natural semantics as a framework, we mechanized a compiler from Mini-ML to a big-step version of the Modern SECD machine. We were interested in showing that in particular, (coinductive) natural semantics can express translations among high-level languages, as well as translations from a high-level language to machine code (a low-level language). Therefore, our compiler is composed of two phases (see Figure 1): translation of Mini-ML with named variables (MiniML) to Mini-ML in de Bruijn notation ($\mathsf{MiniML}^{\mathsf{dB}}$), as an instance of the first one, and generation of code of the (big-step version of the) Modern SECD machine from Mini-ML in de Bruijn notation (Sections 2.3 and 3.3), as an instance of the second one. In this work, we will concentrate on the code generation phase which is the most interesting one.

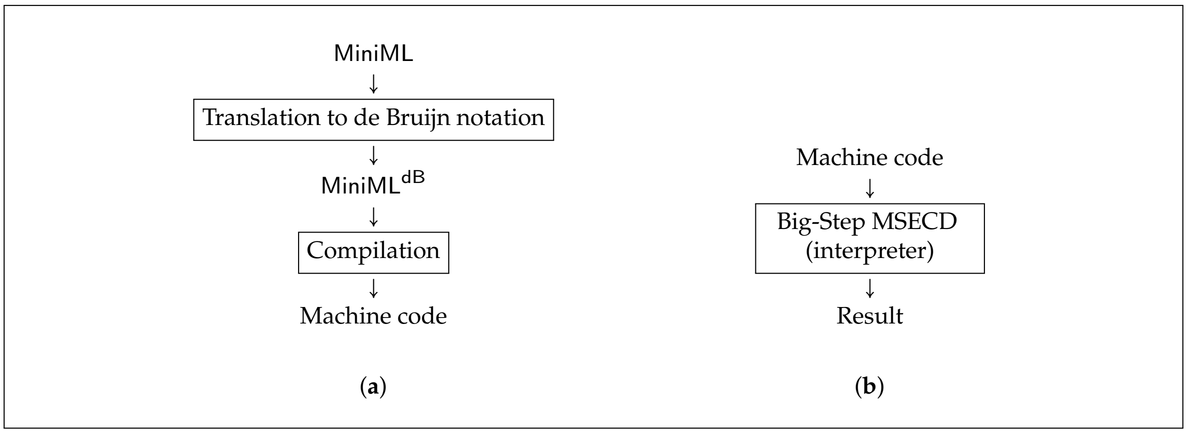

Figure 1. Compiler architecture. (**a**) Compilation; (**b**) Execution.

Traditionally, in the literature small-step semantics is used in the machine. That is why, as an alternative target machine, we offer the original small-step semantics Modern SECD machine (Sections 2.2 and 3.2) extended to support all Mini-ML features, in particular, with native recursion support, mainly to compare it with our solution, in which a big-step machine is used, i.e., our Modern SECD machine big-step version.

1.1. Related Work

Since CompCert C [1–5] project's inception led by Leroy, there has been great progress in the literature dedicated to compiler verification using proof assistants, Coq in particular. In this work, we address specifically functional programming languages verification. In principle, verification of functional programming languages to abstract machines.

An unusual technique exposed by Hardin et al. [16] to carry out the verification of a functional language to an abstract machine is to use small-step semantics, both in the source language and in the abstract machine, together with a decompilation function and a measure to establish correctness. The idea of this technique is to perform a bottom-up simulation in which every machine transition corresponds to zero or one source level reductions. The machine states are mapped back to source level expressions using a decompilation function. More precisely, if from a machine state s a state s' is reached via a machine transition $s \rightarrow s'$, and e is the source language expression corresponding to the state s via decompilation, then there exists an expression e' corresponding to s' via decompilation, such that $e = e'$ or e reduces to e' via source language small-step semantics $e \rightarrow e'$. When a machine performs a transition from a state to another and the decompilation of both states corresponds to the same expression in the source language, the machine performs a *silent transition*. To guarantee that there are not infinitely many silent machine transitions, a measure defined on the machine states is used, i.e., if $s \rightarrow s'$ and the decompilation of s and s' corresponds to the same expression e, then s measure is greater than that of s'.

Gregoire and Leroy [9] and Gregoire [17] use this technique to verify a compiler from a strong reduction lambda calculus to an abstract machine in Coq; more precisely, to verify the correctness of a compiler from the Calculus of Inductive Constructions (CIC) to a variant of the ZAM machine [18] (adapted to support weak symbolic reduction), obtaining a compiler-based verified implementation to evaluate Coq terms. In addition, they show that this compiler-based implementation is more efficient than the original Coq interpreter as expected. More recently, Kunze et al. [19] employ a very similar technique to verify the correctness of a compiler from a call-by-value lambda calculus to an abstract machine in Coq.

However, Leroy [8,14] and Leroy and Grall [15] point out that a correctness proof using this technique is difficult, and also that the definition of a decompilation is complicated, hard to reason about, and hard to extend (especially for optimizing compilation phases). In consequence, they propose a solution based on big-step semantics. In fact, they state that proving semantic preservation for compilers both for terminating and diverging programs using big-step semantics is the original motivation of their work.

The Leroy [14] and Leroy and Grall [15] technique consists of using (coinductive) big-step semantics in the source language but small-step semantics in the machine. In this way, for the termination case, if a source language expression e evaluates to v via big-step semantics $e \Rightarrow v$, then reducing the machine code c via transitive closure of small-step semantics $\overset{+}{\rightarrow}$, takes the machine to a state with v_m at the top of the stack, where c corresponds to e compilation and v_m is the machine value corresponding to v. For the non-termination case, if e diverges using coinductive big-step semantics, then c also diverges in the machine. Leroy and Grall mention that their technique provides a simpler way to prove semantics preservation, in particular for the non-termination case.

Currently, it is well known [14,15,20,21] that big-step semantics are easier and more convenient for compiler correctness proofs, and also for efficient interpreters [21]. Thus, we have on one hand that Leroy and Grall main motivation is to use big-step semantics for compiler correctness proofs, and on the other that big-step semantics has proved to be easier and more convenient for compiler correctness proofs. Our aim is to take big-step semantics to its deepest consequences exploiting it where it has proved to be useful. This is why we propose (coinductive) natural semantics as framework for compiler verification.

(Coinductive) natural semantics as framework for compiler verification in Coq as proposed in this paper is a technique very similar in spirit to that of Leroy and Grall, but going further since not only (coinductive) big-step semantics is used in the source language but also in the target machine (let us recall that Leroy and Grall employ small-step semantics in the machine). Furthermore, to the best of the author's knowledge, it is the first time that coinductive natural semantics is proposed and used to define computations that do not terminate in an abstract machine. In this way, we obtain a fully-based (coinductive) natural semantics technique for a functional language to an abstract machine compiler correctness verification in Coq.

Establishing correctness is even easier, intuitive and simple since natural semantics are also used in the machine. If a source language expression e is evaluated to a value v via source language natural semantics $e \Rightarrow v$, then c is evaluated to a final machine state with v_m at the top of the stack via machine natural semantics $s \vdash c \Rightarrow v_m \cdot s$; where c is the compilation of e via natural semantics $e \Downarrow c$, and v_m is the compilation of v via natural semantics $v \Downarrow v_m$, and s is any machine stack. If e diverges via coinductive natural semantics $e \overset{\infty}{\Rightarrow}$, then c also diverges via machine coinductive natural semantics $s \vdash c \overset{\infty}{\Rightarrow}$. We can note here, how only (coinductive) natural semantics is sufficient to establish correctness; we do not need to use any other distinct formalism.

A potential use of this framework is to take it as basis to verify a conventional compiler to abstract machine implementation of (the core of) a realistic functional language such as OCaml. The official INRIA OCaml implementation comes with two compilers [22], the first one generates code of the ZAM machine, and the second one generates C-- code. We speculate that this framework can also be used as basis to verify the compiler that generates C-- since Dargaye [23] already uses big-step semantics (although not as unifying framework and only tackling terminating computations) to verify a compiler from Mini-ML to Cminor (an early intermediate language of the CompCert C compiler). The idea of generating Cminor (or some other Compcert C intermediate language) code instead of C-- is immediate since, in this way, we can connect the compiler's back-end to CompCert C and obtain as final result verified assembly code. This use takes more relevance if we take into account that Coq itself is an OCaml program (even though some portions of Coq are verified in Coq [24–26], the extracted verified OCaml code will eventually run on an OCaml implementation).

Another line of work is dedicated to systematically derive an abstract machine from a lambda calculus [27–33]. The general idea in these works is from a lambda calculus to carry out a series of transformations until the desired abstract machine is obtained. One of the most exploited transformations in some of these works is refocusing [34], although a great variety of transformations are used. The compilation correctness is a direct consequence of the correctness of the transformations. Some of them, in addition, address Coq formalization [29–33]. The closest works to ours are those which starting from a natural semantics of a lambda calculus derive an abstract machine [27,30]. Specifically, the most similar work in nature to ours is [30]. In [30], the STG machine is derived from natural semantics of a lazy lambda calculus and the derivation is formalized in Coq. However, in [30] only the case for terminating computations is tackled.

In all these works, the emphasis is on the corresponding machine derivation. In contrast, in a functional language implementation the target abstract machine is usually designed by hand and only then (if any) proved correct w.r.t source lambda calculus semantics (see, for example, [18]). Hence, (coinductive) natural semantics as framework as presented in this paper is best suited to verify functional languages implementations (which targets abstract machines), since it assumes that the target machine (and intermediate languages) are given (not to be derived).

Moreover, if for some reason (for example, semantic justification of the target abstract machine) it is considered relevant to systematically derive the target abstract machine from the source calculus, we conjecture that the corresponding derivation can also be carried out in our (coinductive) natural semantics framework. This is because each transformation which leads to the derived machine could be seen as an (intermediate) translation and be defined in natural semantics. Also, the input and output language of each transformation could be seen as an (intermediate) language and its corresponding semantics be defined in natural semantics. Certainly, the derived abstract machine would be a big-step machine.

Other works tackle the verification of a small functional language in Coq, but to the authors' knowledge none of them use (coinductive) natural semantics as unifying framework; instead, they use ad-hoc verifications. For instance, Chlipala [35] offers a compiler from a small impure functional language to an idealized assembly language. He starts from de Bruijn notation and employs natural semantics for the source and target languages, but not to specify the compilation, his effort only cover terminating computations. Benton and Hur [36] deal with the compilation of a small typed functional language to the SECD machine, but they use denotational semantics for the source language and small-step semantics for the target machine. In addition, Benton and Hur employ a biorthogonality step-indexed logical relation to establish correctness. As mentioned before, Dargaye [23] develops a compiler from Mini-ML to C minor, but it is not designed to be a standalone general-purpose Mini-ML implementation. Instead, it was conceived to work only on the code generated by the Coq extraction mechanism. The Coq extraction mechanism generates code of a real-life functional language, by default OCaml, but it is also able to generate Scheme and Haskell code. This is why in Dargaye's work, it only makes sense to cover terminating computations since the Coq's calculus, the Calculus of Inductive Constructions, is strongly normalizing [37], meaning in Coq all computations must terminate. For this reason, all extracted code from Coq should be terminating, while in Coq this property is ensured by Coq's type checker [26]; the code generation translation performed by the Coq extraction mechanism is not verified, although some efforts are conducted in this direction [6,24,25,38–40].

The CertiCoq project [6,40,41] aims to provide a verified extraction pipeline from the core language of Coq, Gallina, to machine language. Therefore, in CertiCoq it also only makes sense to cover terminating computations. This fact is explicitly stated in [6]: '... we can restrict our reasoning to terminating programs since Coq is strongly normalizing. This way we avoid backward simulations (forward simulations proofs are much simpler) and avoid proving preservation of divergence'. Similarly, Savary Bélanger [40] indicates: 'In CertiCoq, we are only concerned with terminating programs: Gallina is strongly normalizing, and our proof of correctness ensures that programs do not acquire non-terminating behaviors along the way'.

Instead of producing machine code directly, CertiCoq generates C light (a CompCert C intermediate language) code. Hence, it uses CompCert C as verified compiler back-end to produce machine language. This way, CertiCoq compiler performs a series of phases from Galllina to Cligth. In CertiCoq, (intermediate) languages semantics and proofs of correctness are based on big-step semantics (for terminating computations). However, big-step semantics is refined with other notions such as step-indexed logical relations and context-based semantics [40,42] to account for additional properties, for instance, compositionality. In addition, the idea of adapting this technique to be useful for general-purpose programming languages is barely mentioned in [40]. Albeit, for this purpose, Savary Bélanger [40] suggests to employ small-step semantics. For their part, Paraskevopoulou and Appel [42], in order to prove closure conversion correctness, they already extend this technique to cover non-termination computations under certain conditions. Closure conversion is a phase performed by CertiCoq.

Our (coinductive) natural semantics framework is best suited to verify usual functional language to abstract machine implementations since it accounts for both terminating and non-terminating computations (total correctness). In addition, it can express terminating and non-terminating computations in an abstract machine. In contrast, by design [6] CertiCoq only covers terminating evaluations on one hand, and on the other it targets C light which is why no abstract machine is used. This situation reflects the fact that our (coinductive) natural semantics framework and CertiCoq pursuit different goals. Although our (coinductive) natural semantics framework is a framework to conduct total correctness compiler verification in Coq, CertiCoq is a verified compiler (from Coq's core calculus to Clight). Hence, it is not an explicit CertiCoq main objective to offer an infrastructure to perform compiler verification [6], even although, the infrastructure and techniques developed to verify the CertiCoq compiler could be adapted to be used to verify other compilers as well.

Step-indexed logical relations as shown by Ahmed [43] serve to establish contextual equivalence between programs. We remark that step-indexed logical relations provide a way to deal with two compiler problems in particular; specifically, compositionality and secure compilation.

In [44], Ahmed and Blume show how to use step-indexed logical relations together with small-step semantics to deal with a notion of secure compilation. Ahmed and Blume demonstrate their method, applying it to a typed closure conversion transformation. Patrignani et al. [45] offer a recent survey of the formal approaches and techniques used in secure compilation. Certainly, this survey includes in particular works that employ step-indexed logical relations. Abate et al. [46] study generalizations of trace-based compiler correctness criteria including some which accounts for secure compilation.

To account for compositionality, Perconti and Ahmed [47] propose the use of a language in which all languages involved in a compilation pipeline can be embedded. Then, using a step-indexed logical relation and small-step semantics compositional compiler correctness is established in terms of the combined language. For their part, Neis et al. [48] introduce parametric inter-language simulations (PILS) as a technique particularly suited to compositional compiler verification for higher-order imperative languages. In particular, they demonstrate their technique with Pilsner, a verified compositional compiler from a ML-like language to an assembly-like language. Patterson and Ahmed [49] provide a framework for expressing different notions of compiler correctness, especially those which consider compiler compositionality.

Dreyer et al. [50], in order to avoid tedious, error-prone and obscuring step-indexed arithmetic, instead of using explicit indices, they propose to 'hide' indices, internalizing them into a logic. The idea is to replace indices with a modal operator, this way obtaining a modal logic which they name LSLR. In particular, this idea is reused in IRIS. IRIS [7,51–53] is a concurrent separation logic framework implemented and verified in Coq. In this regard, Krebbers et al. [53] comment: 'We also show that the step-indexed "later" modality of Iris is an *essential* source of complexity, in that removing it leads to a logical inconsistency'. Recently, Linn Georges et al. [54] formalize a capability machine in IRIS. As Linn Georges et al. [54] point out, capability machines are promising targets for secure compilers. Hence, the idea to extend IRIS to be used as secure compiler framework is imminent; in particular, to verify secure compilers from high-level concurrent languages to capability machines. However, to the authors' knowledge, IRIS has never been used in this manner. A very similar goal is pursued by Cuellar et al. [55] and Cuellar [56] but extending CompCert C. To this end, they introduce the

Concurrent Permission Machine (CPM). Certainly, C (with concurrency) is the source language in these works.

In retrospective, on one hand step-indexed logical relations have proved to be useful, in particular in secure compilation, compiler compositionality and concurrency; on the other hand, natural semantics has proved to be easier and more convenient than other formalisms (for instance, small-step semantics) for compiler correctness proofs. Hence, we speculate that both natural semantics and step-indexed logical relations can be combined in a single formalism that has the best properties of each one of them. In other words, we envisage the ambitious goal of reaching a single formalism that features secure compilation, compositional compilation, concurrency and be simple, easy and intuitive as possible.

Currently, our (coinductive) natural semantics framework does not account for secure compilation, compositionality nor concurrency. However, we conjecture that step-indexed logical relations can be adopted in it to address some or even all these features. The price paid for this effort would be to deal with the known complexity of step-indexed logical relations (although it could be ameliorated, for instance, by internalizing the indices in a natural semantics modal logic). At present, our (coinductive) natural semantics framework is simple, easy and intuitive.

The following are related semantics: coinductive big-step operational semantics [14,15], trace-based coinductive operational semantics [57], pretty-big-step semantics [20] and flag-based big-step semantics [21].

The only one of these works that presents the verification of the correctness of a compiler is Leroy's (an ad-hoc verification). This means that (coinductive) natural semantics is not used in any of them as a unifying framework for the verification of compiler correctness. Specifically, it is not used in the definition of the semantics of the machine (nor in that of its interpreter), it is not used to define the translations, and it is not used (both in the source language and in the target language) to establish, nor to prove the correctness of the translations. What it does, in each of them, is to present a natural semantics with coinduction of a high-level language (which would usually correspond to the source language in a compiler) and it is this aspect that we review next.

Leroy [14] first expresses finite computations with natural semantics 'evaluation' and infinite computations 'divergence' with coinductive natural semantics, separately; this solution is clear and clean. After, he offers an alternative solution in which one finite and infinite computations are expressed in a single coinductive natural semantics 'coevaluation'; however, this semantics does not behave well in the sense that on one hand, there are infinite computations that it is not able to express, and on the other, there are infinite computations that are evaluated to any value v. Nakata and Uustalu [57] remark that this behavior appears accidental and undesired.

Nakata and Uustalu [57] define a coinductive natural semantics of the While language that expresses finite and infinite computations; the careful and ad-hoc design of semantics follows, and within it that of small-step semantics. Additionally, Nakata and Uustalu define an interpreter using the trace monad and they show that it is correct regarding such semantics. Nakata and Uustalu's work [57] is the only one of the related works presented here in which an interpreter is presented.

Charguéraud [20] introduces pretty-big-step semantics, a semantics based on 'coevaluation' of Leroy. Unfortunately, pretty-big-step semantics inherits the not well behavior of 'coevaluation'. In turn, Bach Poulsen and Mosses [21] define flag-based big-step semantics based on pretty-big-step semantics. Unfortunately, flag-based big-step semantics, through pretty-big-step semantics, also inherits the not well behavior of 'coevaluation'.

In this work, we present (coinductive) natural semantics as a framework for the verification of total correctness of compilers in Coq. Once we have a simple, easy, clear, and intuitive solution for this task, we can seek to improve it in the future. In particular, we use a natural parameter in the interpreters to bound the recursion. Recently, Leroy [58] has defined an interpreter of While using the partiality monad in Coq; we plan to adopt the partiality monad in our Mini-ML compiler and in the framework in general to avoid the use of this parameter.

On the other hand, we can seek to reach a single coinductive natural semantics '$\overset{co}{\Rightarrow}$' able to express terminating computations, as well as non-terminating computations. Charguéraud [20] mentions that in principle, this semantics can be used directly to prove total correctness of the translations; however, he points out that the conclusion in the correctness theorem is usually of the form $\exists v'.(v \approx v') \wedge (c \overset{co}{\Rightarrow} v')$, and that the current support of coinduction in Coq only allows using coinductive predicates in the conclusion. In particular, it does not allow using the existential quantifier '$\exists$' or the connective '$\wedge$' when a proof is done by coinduction. Bach Poulsen and Mosses [21] run a similar criticism to the current coinduction support in Coq. Fortunately, our (coinductive) natural semantics is ideal here since, when using (inductive) natural semantics '$\Rightarrow$' to express finite computations and coinductive natural semantics '$\overset{\infty}{\Rightarrow}$' to express infinite computations (separately), the proof of the termination case where the conclusion requires an $\exists$ and an $\wedge$ can be done by induction, whereas, in this way, in the case of non-termination, in the conclusion neither the $\exists$ nor the $\wedge$ is required, only the coinductive predicate $\overset{\infty}{\Rightarrow}$ is used, so it can be proved by coinduction (with the current support of Coq).

Even then, it would be possible to aim at having a single semantics in order to have a more concise definition. If so, the framework could automate the translation from it to the two separated semantics ($\Rightarrow$ and $\overset{\infty}{\Rightarrow}$); also, the framework could establish and prove the equivalence between the first one and the union of the last two semantics. Having arrived at these two semantics, the current results of the framework can be used.

The central problem is that (to the authors' knowledge) to date, there is not a single coinductive natural semantics in the literature that expresses finite and infinite computations, and that it does behave well. The first author, based on Leroy's 'coevaluation', has succeeded in defining a single coinductive natural semantics (of the pure lambda calculus extended with constants) that expresses terminating and non-terminating computations, and that it does behave well in Coq. Also, he has proved the equivalence of this semantics with the union of the two semantics ($\Rightarrow$ and $\overset{\infty}{\Rightarrow}$) that express, respectively, finite and infinite computations separately. Apparently, this result is sound [59] and we plan to present it in future works.

To continue, it would be possible to deal with the problem of decreasing the number of rules necessary in a coinductive natural semantics definition. This is the main goal of pretty-big-step semantics and flag-based big-step semantics. To this end and going further, we envisage that the results in this work and those of pretty-big-step semantics and flag-based big-step semantics (future work and perhaps other works as well) can be integrated in a coinductive natural semantics framework having all the desired properties of each of them. In other words, it is our intention that the resulting coinductive natural semantics framework synthesizes all the major advances in natural semantics.

1.2. Contributions

Our main general and specific contributions are:

- The (coinductive) natural semantics as a framework for the verification of total correctness of compilers in Coq. Such that a working standalone verified compiler, meaning, a compiler sound and complete regarding (coinductive) natural semantics specification and correct regarding semantic preservation of the specified translation can be obtained as a final product
- A systematic method to obtain either a sound or complete interpreter (Sections 2.1 and 3.1) or compiler (Section 2.3.1), as applicable, from a (coinductive) natural semantics specification
- The use of coinductive natural semantics to specify non-terminating computations in an abstract machine (Section 3.3)
- A compiler from Mini-ML to a big-step version of the Modern SECD machine including its total correctness verification in Coq (Sections 2.1, 2.3, 3.1 and 3.3)
- An extended version of the original small-step semantics Modern SECD machine which includes native recursion support (Sections 2.2 and 3.2)

- A big-step version of the Modern SECD machine including its formalization in Coq (Sections 2.3 and 3.3)
- A coinductive natural semantics specification of non-terminating computations of Mini-ML (extended version of Leroy's specification of the pure lambda calculus with constants, Section 3.1)
- A coinductive natural semantics specification of non-terminating computations of the big-step Modern SECD machine (Section 3.3)
- An algorithm to translate from (an abstract representation of) the (coinductive) natural semantics specification of a total correct compiler to its corresponding formalization in Coq (Section 4)

The strategy for the presentation is, first, to tackle the termination case using natural semantics (Section 2), and then the non-termination case using coinductive natural semantics (Section 3). During this work, the method that we use is to present each of the compiler's components together with their corresponding Coq formalization; in this way, it is intended that when our compiler is finished, we will have the necessary intuition behind the algorithm to go from a total correct compiler in abstract to Coq (Section 4). Finally, (Section 5), we present our conclusions.

2. Natural Semantics

We will first tackle, in this section, the case in which the computations are finite (terminating) using natural semantics.

2.1. MiniML$^{\mathsf{dB}}$

To begin with, we introduce the source language, Mini-ML, in de Bruijn notation, which is essentially the pure lambda calculus extended with naturals, Booleans, arithmetic and comparison operators, local definitions, conditionals and native recursion by means of a fixed point operator. Its abstract syntax is the following:

d	$::=$	n	Naturals
	$\mid$	b	Booleans
	$\mid$	$d \star d$ with $\star \in \{+,-,*,=\}$	Primitive operators
	$\mid$	$\underline{i}$	Nameless variables (de Bruijn index)
	$\mid$	`if` d `then` d `else` d	Conditionals
	$\mid$	`let` d `in` d	Local definition
	$\mid$	$\lambda.d$	Abstraction
	$\mid$	$\mu.\lambda.d$	Fixed point
	$\mid$	$d\ d$	Application

Before carrying out the coding of our definitions in Coq, it is important to highlight some of its features. Coq is based on the Calculus of Inductive Constructions, i.e., a lambda calculus with a sophisticated and expressive type system. Since it is a lambda calculus, it can be used as a logic, but also as a programming language, i.e., we can prove propositions, but also write programs. This distinction is made explicit by using the types `Prop` and `Set` respectively. Roughly, it can be said that when a term in Coq has type `Prop`, it is used as logic, and if a term has type `Set`, then it is used as programming language. In fact, this explicit syntactic distinction between `Prop` and `Set` was introduced by the Coq extraction mechanism [60] to distinguish between those terms with computational content and those without it (Paulin-Mohring [60] calls 'Spec' what would later be called 'Set'). In this way, if a term in Coq has a `Set` type, the extraction mechanism can generate a program written in a general-purpose programming language, such as OCaml, related to this term, in contrast to a term with a `Prop` type from which is not possible to extract any program at all.

The abstract syntax of MiniML$^{\mathsf{dB}}$ can be coded in Coq as first order abstract syntax using an `Inductive` definition with type `Set` as follows:

```
Inductive MML_dB_exp: Set :=
| Const_dB : nat → MML_dB_exp
| ...
| Letm_dB: MML_dB_exp → MML_dB_exp → MML_dB_exp
| ...
| Lam_dB: MML_dB_exp → MML_dB_exp
| Mu_dB: MML_dB_exp → MML_dB_exp
| App_dB: MML_dB_exp → MML_dB_exp → MML_dB_exp.
```

For conciseness, throughout this article, we will show only the essentials parts of the formalization. The full formalization can be consulted in [61].

Hence, the Coq extraction mechanism can be used with the following command:

```
Extraction MML_dB_exp.
```

which gives as a result:

```
type mML_dB_exp =
| Const_dB of nat
| ...
| Letm_dB of mML_dB_exp * mML_dB_exp
| ...
| Lam_dB of mML_dB_exp
| Mu_dB of mML_dB_exp
| App_dB of mML_dB_exp * mML_dB_exp
```

This way, we can notice how an `Inductive` definition with type `Set` in Coq corresponds to an ADT in OCaml, in this case to the abstract syntax of MiniML$^{\mathsf{dB}}$ written in OCaml.

To define the natural semantics of MiniML$^{\mathsf{dB}}$, we first need to define its values by means of environments and closures.

v	$::=$	n	Numbers	Ω	$::=$	$[]$	Empty environment
	$\mid$	b	Booleans		$\mid$	$v \cdot \Omega$	
	$\mid$	$(\lambda.d)[\Omega]$	Closures				
	$\mid$	$(\mu.\lambda.d)[\Omega]$	Recursive closures				

The environments serve as (implicit) associations from variables (represented by de Bruijn indices) to values. In this manner, as expressed by the predicate $\Omega \vdash \underline{i} \longmapsto v$, the value of a variable represented by the index $\underline{i}$ is at the i^{th} position in the environment (a sequence of values).

$$\frac{}{v \cdot \Omega \vdash \underline{0} \longmapsto v} \qquad \frac{\Omega \vdash \underline{i} \longmapsto v}{w \cdot \Omega \vdash \underline{S\ i} \longmapsto v}$$

The natural semantics of MiniML$^{\mathsf{dB}}$ is inductively defined by the predicate:

$$\Omega \vdash d \Rightarrow v$$

which can be read as follows: in the environment Ω, the expression d is evaluated to the value v.

The Ω environment is supposed to contain the value of the free variables in d. The natural semantics of MiniML$^{\mathsf{dB}}$ is defined as follows:

$$\frac{}{\Omega \vdash n \Rightarrow n} \qquad \frac{}{\Omega \vdash b \Rightarrow b} \qquad \frac{\Omega \vdash d_1 \Rightarrow n_1 \qquad \Omega \vdash d_2 \Rightarrow n_2}{\Omega \vdash d_1 \star d_2 \Rightarrow n_1 \star n_2}$$

$$\frac{\Omega \vdash \underline{i} \longmapsto v}{\Omega \vdash \underline{i} \Rightarrow v} \qquad \frac{\Omega \vdash d_1 \Rightarrow v_1 \qquad v_1 \cdot \Omega \vdash d_2 \Rightarrow v}{\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \Rightarrow v}$$

$$\frac{\Omega \vdash d_1 \Rightarrow \mathit{true} \qquad \Omega \vdash d_2 \Rightarrow v}{\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Rightarrow v} \qquad \frac{\Omega \vdash d_1 \Rightarrow \mathit{false} \qquad \Omega \vdash d_3 \Rightarrow v}{\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Rightarrow v}$$

$$\frac{}{\Omega \vdash \lambda.d \Rightarrow (\lambda.d)[\Omega]} \qquad \frac{}{\Omega \vdash \mu.\lambda.d \Rightarrow (\mu.\lambda.d)[\Omega]}$$

$$\frac{\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1] \qquad \Omega \vdash d_2 \Rightarrow v_2 \qquad v_2 \cdot \Omega_1 \vdash d \Rightarrow v}{\Omega \vdash d_1\ d_2 \Rightarrow v}$$

$$\frac{\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1] \qquad \Omega \vdash d_2 \Rightarrow v_2 \qquad v_2 \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \Rightarrow v}{\Omega \vdash d_1\ d_2 \Rightarrow v}$$

A natural semantics definition can be seen as an inductive logical proposition; hence, it can be encoded in Coq as an `Inductive` definition with type `Prop`. This way the MiniML$^{\mathsf{dB}}$ semantics can be written as follows:

```
Inductive MML_dB_NS : MML_dB_env → MML_dB_exp → MML_dB_val → Prop :=
| MML_dB_NS_Const: ∀ O: MML_dB_env, ∀ n: nat,
    MML_dB_NS O (Const_dB n) (Num_dB n)
| ...
| MML_dB_NS_Let: ∀ O: MML_dB_env, ∀ d1 d2: MML_dB_exp, ∀ v1 v2: MML_dB_val,
    MML_dB_NS O d1 v1 →
    MML_dB_NS (v1:: O) d2 v2 →
    MML_dB_NS O (Letm_dB d1 d2) v2
| ...
| MML_dB_NS_Lam: ∀ O: MML_dB_env, ∀ d: MML_dB_exp,
    MML_dB_NS O (Lam_dB d) (Clos_dB d O)

| MML_dB_NS_Mu: ∀ O: MML_dB_env, ∀ d: MML_dB_exp,
    MML_dB_NS O (Mu_dB d) (Closr_dB d O)

| MML_dB_NS_App: ∀ O O1: MML_dB_env, ∀ d1 d2 d: MML_dB_exp, ∀ v v2: MML_dB_val,
    MML_dB_NS O d1 (Clos_dB d O1) →
    MML_dB_NS O d2 v2 →
    MML_dB_NS (v2:: O1) d v →
    MML_dB_NS O (App_dB d1 d2) v

| MML_dB_NS_Appr: ∀ O O1: MML_dB_env, ∀ d1 d2 d: MML_dB_exp, ∀ v v2: MML_dB_val,
    MML_dB_NS O d1 (Closr_dB d O1) →
    MML_dB_NS O d2 v2 →
    MML_dB_NS (v2::( Closr_dB d O1):: O1) d v →
    MML_dB_NS O (App_dB d1 d2) v.
```

A natural semantics definition, in general, is a relation; therefore, no determinism is the general case. Then, in case a definition of this kind is deterministic, a lemma that expresses this property must be formally established in the same way we do it for the MiniML$^{\mathsf{dB}}$ semantics:

```
Lemma MML_dB_NS_deterministic:
∀ O,  ∀ d,  ∀ v1,  MML_dB_NS O  d  v1 →
                ∀ v2,  MML_dB_NS O  d  v2 →
                         v1  =  v2.
```

If a relation is deterministic, then it can be stated as a function. Consequently, it can be encoded as function in a programming language. In particular, in Coq, we can write a recursive function employing `Fixpoint`. For instance, the function corresponding to MiniML^{dB} natural semantics is written as follows:

```
Fixpoint MML_dB_NS_interpreter (depthR:nat )( O: MML_dB_env) ( d: MML_dB_exp)
:  option MML_dB_val :=
match depthR with
| 0  ⇒  None
| S  m  ⇒  match d with
           | Const_dB n  ⇒  Some (Num_dB n)
           | ...
           | Letm_dB d1  d2  ⇒
             match ( MML_dB_NS_interpreter m O  d1)  with
               | Some  v1  ⇒  MML_dB_NS_interpreter m (v1::O)  d2
               | _  ⇒  None
             end
           | ...
           | Lam_dB  d  ⇒  Some (Clos_dB d  O)
           | Mu_dB  d  ⇒  Some (Closr_dB d  O)
           | App_dB d1  d2  ⇒
             match ( MML_dB_NS_interpreter m O  d1)  with
               | Some  ( Clos_dB  d  O1)  ⇒
                 match MML_dB_NS_interpreter m O  d2  with
                   | Some  v2  ⇒  MML_dB_NS_interpreter m (v2::O1)  d
                   | _  ⇒  None
                 end
               | Some  ( Closr_dB d  O1)  ⇒
                 match MML_dB_NS_interpreter m O  d2  with
                   | Some  v2  ⇒  MML_dB_NS_interpreter m (v2::( Closr_dB d  O1)::  O1)  d
                   | _  ⇒  None
                 end
               | _  ⇒  None
             end
         end
end.
```

Please note that this function is actually an interpreter. To guarantee termination, we added the natural parameter `depthR` which indicates the recursion depth (`depthR` is also called '`fuel`' by some authors, see, for example [24,25,62,63]). This is because the CIC is strongly normalizing [37], which means, from a programming-language perspective, that all the calculations must terminate. In Coq, this means that all functions must be total and terminating.

From a verification perspective, a logical proposition serves as a formal specification that a program must comply. In this case, the logical proposition is the `Inductive` definition in `Prop` whereas the program is the interpreter in `Set`. Then, to verify that our interpreter is sound with respect to the MiniML^{dB} natural semantics, we must prove the following lemma:

```
Lemma MML_dB_NS_interpreter_soundness:
∀ O,  ∀ d,  ∀ v,  MML_dB_NS O d v →
          ∃ n,  MML_dB_NS_interpreter n O d = Some v.
```

Conversely, i.e., to verify that our interpreter is complete with respect to MiniML^{dB} natural semantics, we must prove this lemma:

```
Lemma MML_dB_NS_interpreter_completeness: ∀ n, ∀ O,  ∀ d,  ∀ v,
MML_dB_NS_interpreter n O d = Some v →
MML_dB_NS O d v.
```

Now we write factorial of 5 in MiniML^{dB} as a program example:

```
Definition example :=
(App_dB (Mu_dB (If_dB (Eq_dB (Var_dB 0)  (Const_dB 0))
         (Const_dB 1)
         (Times_dB (Var_dB 0)
            (App_dB (Var_dB 1)
                    (Minus_dB (Var_dB 0)  (Const_dB 1))))))    (Const_dB 5)).
```

then, we can evaluate it in our MiniML^{dB} interpreter, inside Coq, as follows:

```
Compute (MML_dB_NS_interpreter 19 nil example).
```

obtaining just the expected result:

```
= Some (Num_dB 120)  :  option MML_dB_val
```

Next, we can use the extraction mechanism as follows:

```
Extraction MML_dB_NS_interpreter.
```

so, in this way, we obtain a verified interpreter, sound and complete with respect to MiniML^{dB} natural semantics in OCaml, ready to be used in real life.

```
let rec mML_dB_NS_interpreter depthR o d =
  match depthR with
   | O → None
   | S m →
     (match d with
       | Const_dB n → Some (Num_dB n)
       | ...
       | Letm_dB (d1, d2) →
         (match mML_dB_NS_interpreter m o d1 with
           | Some v1 → mML_dB_NS_interpreter m (Cons (v1, o)) d2
           | None → None)
       | ...
       | Lam_dB d0 → Some (Clos_dB (d0, o))
       | Mu_dB d0 → Some (Closr_dB (d0, o))
       | App_dB (d1, d2) →
```

```
        (match mML_dB_NS_interpreter m o d1 with
         | Some m0 →
           (match m0 with
            | Clos_dB (d0, o1) →
              (match mML_dB_NS_interpreter m o d2 with
               | Some v2 → mML_dB_NS_interpreter m (Cons (v2, o1)) d0
               | None → None)
            | Closr_dB (d0, o1) →
              (match mML_dB_NS_interpreter m o d2 with
               | Some v2 →
                 mML_dB_NS_interpreter m (Cons (v2, (Cons (( Closr_dB (d0, o1)),
                   o1)))) d0
               | None → None)
            | _ → None)
         | None → None))
```

The reader may question the 'double' task of maintaining both definitions, `Prop` and `Set`. On one hand, if we stay in the logical part, in `Prop`, a verified compiler cannot be obtained to be used in real life, while on the other hand, definitions using the `Set` type forces us to work with total terminating functions.

Let us recall that natural semantics is, in general, inherently relational and, non-deterministic; therefore, to write a natural semantics definition as a function we must ensure that it is total and deterministic. Although for some particular cases this is true, we think that if a natural semantics definition is written directly as a function, the essence of natural semantics vanishes.

Also, Coq automatically generates inductive principles from inductive definitions, which is not the case for functions. These induction principles are useful as they can be used through the `induction` tactic while doing a proof. In some scenarios, this is an advantage, especially, of course, when a proof is done by induction.

Regarding the remarks above, we give definitions in `Prop` to retain the essence of natural semantics and to take advantage of the inductive principles generated by Coq. Also, we give the corresponding definitions in `Set`, mainly to obtain verified implementations.

2.2. Modern SECD Machine

Leroy [8,14] introduces the Modern SECD, a machine based on Landin's SECD [64] with two main differences: the first one is that it does not use a Dump; instead, it makes use of frames in the stack to support function calls; the second one is that it uses de Bruijn indices to access the environment.

The original Modern SECD only offers natural constants, local definitions, abstraction and application support. Due to this, we offer an extended version of the MSECD to support Booleans, arithmetic and comparison operators, conditionals, and native recursion by means of recursive closures. It is worth mentioning that we made the conditionals support based on Henderson's SECD presentation [65].

The instructions of the extended MSECD are the following:

i	::=	$\mathsf{IConst}\ n$	Push the natural n
	$\mid$	$\mathsf{IConstb}\ b$	Push the Boolean b
	$\mid$	IAdd	Perform an addition
	$\mid$	ISub	Perform a subtraction
	$\mid$	IMul	Perform a multiplication
	$\mid$	IEq	Perform an equality comparison
	$\mid$	$\mathsf{IAcc}\ \underline{i}$	Push the value of the variable number (de Bruijn index) $\underline{i}$
	$\mid$	$\mathsf{ISel}\ c\ c$	Select a conditional branch
	$\mid$	IJoin	Rejoin the main control (return from conditional)
	$\mid$	ILet	Add the value of a local definition variable to the environment
	$\mid$	$\mathsf{IEndLet}$	Remove the value of a local definition variable from the environment
	$\mid$	$\mathsf{IClos}\ c$	Push a closure with code c
	$\mid$	$\mathsf{IClos}_{rec}\ c$	Push a recursive closure with code c
	$\mid$	IApp	Perform a function application
	$\mid$	IRet	Return from function

Notice the distinction between 'i' and '$\underline{i}$', the former denotes a machine instruction, whereas the latter denotes a de Bruijn index.

The code is defined as an instruction sequence:

c	::=	$[]$	Empty code
	$\mid$	$i \cdot c$	

The values of the machine and its environment are defined as follows:

v_m	::=	n	Naturals	Δ	::=	$[]$	Empty environment
	$\mid$	b	Booleans		$\mid$	$v_m \cdot \Delta$	
	$\mid$	$c[\Delta]$	Closures				
	$\mid$	$c[\Delta]_{rec}$	Recursive closures				

Besides these values, the frames should be able to be stored in the stack. Therefore, the stack values and the stack of the machine are defined as follows:

v_s	::=	v_m	Machine values	s	::=	$[]$	Empty stack
	$\mid$	(c, Δ)	Stack Frames		$\mid$	$v_s \cdot \Delta$	

A MSECD machine *configuration* is a triplet (c, Δ, s) where c is a code, Δ is a machine environment, and s is a stack.

The MSECD small-step semantics is a transition relation from a configuration (c_i, Δ_i, s_i) to the next $(c_{i+1}, \Delta_{i+1}, s_{i+1})$ denoted by: $(c_i, \Delta_i, s_i) \rightarrow (c_{i+1}, \Delta_{i+1}, s_{i+1})$.

Next, we present the MSECD small-step semantics, shown in Table 1.

Table 1. MSECD small-step semantics.

Current			Next		
Code	**Environment**	**Stack**	**Code**	**Environment**	**Stack**
$(\mathsf{IConst}\ n)\cdot c$	Δ	s	c	Δ	$n\cdot s$
$(\mathsf{IConstb}\ b)\cdot c$	Δ	s	c	Δ	$b\cdot s$
$\mathsf{IAdd}\cdot c$	Δ	$n_2\cdot n_1\cdot s$	c	Δ	$n_1+n_2\cdot s$
$\mathsf{ISub}\cdot c$	Δ	$n_2\cdot n_1\cdot s$	c	Δ	$n_1-n_2\cdot s$
$\mathsf{IMul}\cdot c$	Δ	$n_2\cdot n_1\cdot s$	c	Δ	$n_1*n_2\cdot s$
$\mathsf{IEq}\cdot c$	Δ	$n_2\cdot n_1\cdot s$	c	Δ	$n_1=n_2\cdot s$
$(\mathsf{IAcc}\ \underline{i})\cdot c$	$[v_0,\ldots,v_i,\ldots,v_n]=\Delta$	s	c	Δ	$v_i\cdot s$
$\mathsf{ILet}\cdot c$	Δ	$v\cdot s$	c	$v\cdot\Delta$	s
$\mathsf{IEndLet}\cdot c$	$v\cdot\Delta$	s	c	Δ	s
$(\mathsf{ISel}\ c_1\ c_2)\cdot c$	Δ	$true\cdot s$	c_1	Δ	$(c,[])\cdot s$
$(\mathsf{ISel}\ c_1\ c_2)\cdot c$	Δ	$false\cdot s$	c_2	Δ	$(c,[])\cdot s$
$\mathsf{IJoin}\cdot c$	Δ	$v\cdot(c_b,[])\cdot s$	c_b	Δ	$v\cdot s$
$(\mathsf{IClos}\ c_1)\cdot c$	Δ	s	c	Δ	$c_1[\Delta]\cdot s$
$(\mathsf{IClos}_{rec}\ c_1)\cdot c$	Δ	s	c	Δ	$c_1[\Delta]_{rec}\cdot s$
$\mathsf{IApp}\cdot c$	Δ	$v\cdot c_1[\Delta_1]\cdot s$	c_1	$v\cdot\Delta_1$	$(c,\Delta)\cdot s$
$\mathsf{IApp}\cdot c$	Δ	$v\cdot c_1[\Delta_1]_{rec}\cdot s$	c_1	$v\cdot c_1[\Delta_1]_{rec}\cdot\Delta_1$	$(c,\Delta)\cdot s$
$\mathsf{IRet}\cdot c$	Δ	$v\cdot(c_1,\Delta_1)\cdot s$	c_1	Δ_1	$v\cdot s$

To codify this semantics in Coq, we write:

```
Inductive MSECD_SS: conf → conf → Prop :=
| MSECD_SS_IConst: ∀ n: nat, ∀ c: code, ∀ D: env, ∀ s: stack,
    MSECD_SS (IConst n:: c, D, s) (c, D, Val (MInt n):: s)
| ...

| MSECD_SS_ILet: ∀ c: code, ∀ D: env, ∀ v: val, ∀ s: stack,
    MSECD_SS (ILet:: c, D, Val v:: s) (c, v:: D, s)

| MSECD_SS_IEndLet: ∀ c: code, ∀ D: env, ∀ v: val, ∀ s: stack,
    MSECD_SS (IEndLet:: c, v:: D, s) (c, D, s)
| ...

| MSECD_SS_IClos: ∀ cc c: code, ∀ D: env, ∀ s: stack,
    MSECD_SS (IClos cc:: c, D, s) (c, D, Val (MClos cc D):: s)

| MSECD_SS_IClosr: ∀ cc c: code, ∀ D: env, ∀ s: stack,
    MSECD_SS (IClosr cc:: c, D, s) (c, D, Val (MClosr cc D):: s)

| MSECD_SS_IApp: ∀ c c1: code, ∀ D D1: env, ∀ s: stack, ∀ v: val,
    MSECD_SS (IApp:: c, D, Val v:: Val (MClos c1 D1):: s) (c1, v:: D1, SFrame c D:: s)

| MSECD_SS_IAppr:∀ c c1: code, ∀ D D1: env, ∀ s: stack, ∀ v: val,
    MSECD_SS (IApp:: c, D, Val v:: Val (MClosr c1 D1):: s) (c1, v::( MClosr c1 D1):: D1, SFrame c D:: s)

| MSECD_SS_IReturn: ∀ c c1: code, ∀ D D1: env, ∀ v: val, ∀ s: stack,
    MSECD_SS (IRet:: c, D, Val v:: SFrame c1 D1 :: s) (c1, D1, Val v:: s).
```

Let m_1, m_2 and m_3 be MSECD machine configurations, the transitive closure of the small-step semantics is defined inductively as follows:

$$\frac{m_1 \to m_2}{m_1 \overset{+}{\to} m_2} \qquad\qquad \frac{m_1 \to m_2 \qquad m_2 \overset{+}{\to} m_3}{m_1 \overset{+}{\to} m_3}$$

In Coq, this transitive closure is written as follows:

```
Inductive TC_MSECD_SS: conf → conf → Prop :=
| TC_MSECD_SS_SS:
    ∀ m1 m2, MSECD_SS m1 m2 →
              TC_MSECD_SS m1 m2

| TC_MSECD_SS_Transitivity:
    ∀ m1 m2, MSECD_SS m1 m2 →
       ∀ m3, TC_MSECD_SS m2 m3 →
              TC_MSECD_SS m1 m3.
```

2.2.1. Compilation

Leroy [14] defines the compilation from the pure lambda calculus extended with constants to MSECD machine code as a function. Here, we extend his work to all the MiniML$^{\mathsf{dB}}$ language constructs:

$$\begin{array}{rcl}
[\![n]\!] & = & \mathsf{IConst}\ n \\
[\![b]\!] & = & \mathsf{IConstb}\ b \\
[\![d_1 \star d_2]\!] & = & [\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IOp} \\
[\![\underline{i}]\!] & = & \mathsf{IAcc}\ \underline{i} \\
[\![\texttt{let}\ d_1\ \texttt{in}\ d_2]\!] & = & [\![d_1]\!] \cdot \mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \\
[\![\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3]\!] & = & [\![d_1]\!] \cdot \mathsf{ISel}\ ([\![d_2]\!] \cdot \mathsf{IJoin})\ ([\![d_3]\!] \cdot \mathsf{IJoin}) \\
[\![\lambda.d]\!] & = & \mathsf{IClos}\ ([\![d]\!] \cdot \mathsf{IRet}) \\
[\![\mu.\lambda.d]\!] & = & \mathsf{IClos}_{rec}\ ([\![d]\!] \cdot \mathsf{IRet}) \\
[\![d_1\ d_2]\!] & = & [\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp}
\end{array}$$

2.2.2. Correctness

The correctness of this compilation can be established by semantic preservation for which it is necessary to extend the compilation to values and environments of the machine, as shown below:

$$[\![n]\!] = n \qquad [\![b]\!] = b \qquad [\![(\lambda.d)[\Omega]]\!] = ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega]\!]]$$

$$[\![(\mu.\lambda.d)[\Omega]]\!] = ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega]\!]]_{rec} \qquad [\![v_1 \dots v_n]\!] = [\![v_1]\!] \dots [\![v_n]\!]$$

In this way, if an expression d is evaluated to a value v in an Ω environment, it is expected that its compilation $[\![d]\!]$ is evaluated to $[\![v]\!]$ in the $[\![\Omega]\!]$ environment. However, to prove this result, it is necessary to strengthen the hypothesis (we will see, in Section 2.3.2 that this is not necessary when natural semantics is used). Here, to strengthen the hypothesis is to concatenate any code c at the end of compilation $[\![d]\!]$, so when the evaluation of $[\![d]\!]$ finishes, it is expected that $[\![v]\!]$ is at the top of the stack, and the code c remains to evaluate. This is formally expressed in the following theorem formulated by Leroy [14]:

Theorem 1. *If* $\Omega \vdash d \Rightarrow v$*, then* $([\![d]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow{+} (c, [\![\Omega]\!], [\![v]\!] \cdot s)$ *for all codes* c *and stacks* s*.*

Proof outline. The proof is conducted by induction on the derivation of $\Omega \vdash d \Rightarrow v$. The base cases where d is a natural, a Boolean, a nameless variable, an abstraction, or a fixed point (recursive abstraction) are straightforward since the corresponding $[\![d]\!]$ compilation is a single machine instruction whose evaluation is performed by a single machine step $\rightarrow$ that mimics the MiniML$^{\mathsf{dB}}$ d evaluation.

For the inductive cases where d is an arithmetic or comparison expression, a conditional, a local definition, or an application the proof follows the structure of the derivation $\Omega \vdash d \Rightarrow v$. The key idea is to use the $\xrightarrow{+}$ transitivity together with the induction hypothesis, while evaluating the intermingled single instructions which appear in $[\![d]\!]$ by performing the corresponding $\rightarrow$ machine step. □

The complete proof can be consulted in Appendix A. This theorem is written in Coq as follows:

```
Theorem compile_eval:
∀ O, ∀ d, ∀ v, MML_dB_NS O d v →
    ∀ c, ∀ s, TC_MSECD_SS ((compile d)++c, compile_env O,s)
                          (c, compile_env O, (Val (compile_value v):: s)).
```

2.3. Big-Step MSECD Machine

This section introduces our big-step version of the Modern SECD machine. This machine is strongly based on our extended version of the original small-step semantics MSECD. Unlike the small-step MSECD, due to the high-level of abstraction of natural semantics, it is not necessary to use stack frames at all, and therefore the return instructions are also unnecessary (IRet which works for returning from a function call, nor IJoin that works for returning from a conditional). Having said this, we can affirm that the use of natural semantics directly impacts the machine design, specifically the machine's components.

The instructions of the big-step MSECD machine, as well as its code, are the following:

i	::=	IConst n	Naturals
	|	IConstb b	Booleans
	|	IAdd	Addition
	|	ISub	Subtraction
	|	IMul	Multiplication
	|	IEq	Equality comparison
	|	IAcc $\underline{i}$	Variable value access (de Bruijn index)
	|	ISel c c	Conditionals
	|	ILet	Local definitions
	|	IEndLet	
	|	IClos c	Abstraction
	|	IClos$_{rec}$ c	Recursion abstraction
	|	IApp	Application

c	::=	$[]$	Empty code
	|	$i \cdot c$	

The values and environments of the machine are defined as follows:

v_m	::=	n	Naturals
	|	b	Booleans
	|	$c[\Delta]$	Closures
	|	$c[\Delta]_{rec}$	Recursive closures

Δ	::=	$[]$	Empty environment
	|	$v_m \cdot \Delta$	

Given the fact that the frames disappear, it is not necessary to define stack values. Due to this, the stack directly becomes a sequence of machine values:

s	::=	$[]$	Empty stack
	|	$v_m \cdot s$	

The predicate $\Delta \vdash \underline{i} \longmapsto v$ expresses that v is the value of the variable represented by the de Bruijn index $\underline{i}$ in the machine environment Δ.

$$\frac{}{v \cdot \Delta \vdash \underline{0} \longmapsto v} \qquad\qquad \frac{\Delta \vdash \underline{i} \longmapsto v}{w \cdot \Delta \vdash \underline{S\ i} \longmapsto v}$$

A *state* is a pair (Δ, s) where Δ is a machine environment and s a stack.

The machine natural semantics is defined by the following two mutually dependent predicates:

$$\Delta, s \vdash c \Rightarrow (\Delta_f, s_f) \qquad\qquad \Delta_1, s_1 \vdash i \Rightarrow (\Delta_2, s_2)$$

the first one for machine code, which can be read as follows: if the machine is in a state (Δ, s), and a code c is given, evaluating c takes it to the state (Δ_f, s_f). The second one for instructions, which can be read as follows: if the machine is in a state (Δ_1, s_1) and an instruction i is given, evaluating i takes it to the state (Δ_2, s_2). However, the entry point for the semantics should be the predicate for code.

The environment Δ is supposed to contain the value of the free variables (represented by $\mathsf{IAcc}\ \underline{i}$ instructions) in c, whereas the environment Δ_1 are supposed to contain the value of the free variable in i (if i is an instruction $\mathsf{IAcc}\ \underline{i}$). The natural semantics of the machine is the following:

$$\frac{}{\Delta, s \vdash [\,] \Rightarrow (\Delta, s)} \qquad\qquad \frac{\Delta, s \vdash i \Rightarrow (\Delta_1, s_1) \qquad \Delta_1, s_1 \vdash c \Rightarrow (\Delta_2, s_2)}{\Delta, s \vdash i \cdot c \Rightarrow (\Delta_2, s_2)}$$

$$\frac{}{\Delta, s \vdash \mathsf{IConst}\ n \Rightarrow (\Delta, n \cdot s)} \qquad\qquad \frac{}{\Delta, s \vdash \mathsf{IConstb}\ b \Rightarrow (\Delta, b \cdot s)}$$

$$\frac{}{\Delta, n_2 \cdot n_1 \cdot s \vdash \mathsf{IOp} \ \Rightarrow (\Delta, n_1 \star n_2 \cdot s)}\ \textsf{IOp} \in \{\textsf{IAdd}, \textsf{ISub}, \textsf{IMul}, \textsf{IEq}\},\ \star \in \{+, -, *, =\} \text{ resp.}$$

$$\frac{\Delta \vdash \underline{i} \longmapsto v}{\Delta, s \vdash \mathsf{IAcc}\ \underline{i} \Rightarrow (\Delta, v \cdot s)}$$

$$\frac{\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1)}{\Delta, \mathit{true} \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Rightarrow (\Delta_1, s_1)} \qquad\qquad \frac{\Delta, s \vdash c_2 \Rightarrow (\Delta_1, s_1)}{\Delta, \mathit{false} \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \Rightarrow (\Delta_1, s_1)}$$

$$\frac{}{\Delta, v \cdot s \vdash \mathsf{ILet} \ \Rightarrow (v \cdot \Delta, s)} \qquad\qquad \frac{}{v \cdot \Delta, s \vdash \mathsf{IEndLet} \ \Rightarrow (\Delta, s)}$$

$$\frac{}{\Delta, s \vdash \mathsf{IClos}\ c \Rightarrow (\Delta, c[\Delta] \cdot s)} \qquad\qquad \frac{}{\Delta, s \vdash \mathsf{IClos}_{rec}\ c \Rightarrow (\Delta, c[\Delta]_{rec} \cdot s)}$$

$$\frac{v \cdot \Delta_1, s \vdash c \Rightarrow (\Delta_2, v_1 \cdot s_1)}{\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \ \Rightarrow (\Delta, v_1 \cdot s_1)} \qquad\qquad \frac{v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \Rightarrow (\Delta_2, v_1 \cdot s_1)}{\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \ \Rightarrow (\Delta, v_1 \cdot s_1)}$$

The coding of the natural semantics of the machine is similar to that of the natural semantics of $\mathsf{MiniML}^{\mathsf{dB}}$ (only, it is necessary to use a mutually dependent definition in Coq, in correspondence with the mutually dependent predicates of the machine natural semantics). The same applies for the interpreter and its respective lemmas of soundness and completeness regarding the machine natural semantics. The formalization details can be consulted at [61].

We highlight that the machine natural semantics has the property enunciated in the following lemma:

Lemma 1. *Let Δ, Δ_1, Δ_2 be machine environments; s, s_1, s_2 stacks; c_1, c_2 machine codes. If*

$$\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1), \qquad \Delta_1, s_1 \vdash c_2 \Rightarrow (\Delta_2, s_2)$$

then

$$\Delta, s \vdash c_1 \cdot c_2 \Rightarrow (\Delta_2, s_2)$$

Proof outline. By induction on the derivation $\Delta, s \vdash c_1 \Rrightarrow (\Delta, s_1)$. The base case is when c_1 is the empty code $c_1 = []$ which follows simply by hypothesis. The inductive case is when c_1 is not empty $c_1 \neq []$ which is proved by applying the induction hypothesis and by $\Rrightarrow$ definition. □

This lemma is useful to prove compilation correctness (Section 2.3.2). A detailed proof can be seen in Appendix A.

2.3.1. Compilation

Using natural semantics, the compilation from $\mathsf{MiniML}^{\mathsf{dB}}$ to code of the big-step Modern SECD machine is defined by the following predicate:

$$d \Downarrow c$$

meaning, the $\mathsf{MiniML}^{\mathsf{dB}}$ expression d is compiled into the machine code c.

$$\frac{}{n \Downarrow \mathsf{IConst}\ n} \qquad \frac{}{b \Downarrow \mathsf{IConstb}\ b} \qquad \frac{d_1 \Downarrow c_1 \qquad d_2 \Downarrow c_2}{d_1 \star d_2 \Downarrow c_1 \cdot c_2 \cdot \mathsf{IOp}}\ \star \in \{+,-,*,=\}, \mathbf{IOp} \in \{\mathbf{IAdd}, \mathbf{ISub}, \mathbf{IMul}, \mathbf{IEq}\} \text{ resp.}$$

$$\frac{}{\underline{i} \Downarrow \mathsf{IAcc}\ \underline{i}} \qquad \frac{d_1 \Downarrow c_1 \qquad d_2 \Downarrow c_2}{\texttt{let}\ d_1\ \texttt{in}\ d_2 \Downarrow c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet}} \qquad \frac{d_1 \Downarrow c_1 \qquad d_2 \Downarrow c_2 \qquad d_3 \Downarrow c_3}{\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c_1 \cdot \mathsf{ISel}\ c_2\ c_3}$$

$$\frac{d \Downarrow c}{\lambda.d \Downarrow \mathsf{IClos}\ c} \qquad \frac{d \Downarrow c}{\mu.\lambda.d \Downarrow \mathsf{IClos}_{rec}\ c} \qquad \frac{d_1 \Downarrow c_1 \qquad d_2 \Downarrow c_2}{d_1\ d_2 \Downarrow c_1 \cdot c_2 \cdot \mathsf{IApp}}$$

Regarding the Coq encoding of this compilation, it is analogous to the $\mathsf{MiniML}^{\mathsf{dB}}$ semantics, meaning, it is done with an `Inductive` definition with `Prop` type:

```
Inductive Compilation_NS: MML_dB_exp → code → Prop :=
| Compilation_NS_Const: ∀ n: nat,
              Compilation_NS (Const_dB n) (IConst n:: nil)
| ...

| Compilation_NS_Let:
    ∀ d1, ∀ c1, Compilation_NS d1 c1 →
    ∀ d2, ∀ c2, Compilation_NS d2 c2 →
              Compilation_NS (Letm_dB d1 d2) (c1 ++ILet:: c2  ++IEndLet:: nil)
| ...

| Compilation_NS_Lam:
      ∀ d, ∀ c, Compilation_NS d c →
              Compilation_NS (Lam_dB d) (IClos c:: nil)

| Compilation_NS_Mu:
      ∀ d, ∀ c, Compilation_NS d c →
              Compilation_NS (Mu_dB d) (IClosr c:: nil)

| Compilation_NS_App:
    ∀ d1, ∀ c1, Compilation_NS d1 c1 →
    ∀ d2, ∀ c2, Compilation_NS d2 c2 →
              Compilation_NS (App_dB d1 d2) (c1 ++c2 ++IApp:: nil).
```

and the corresponding function is written as follows:

```
Fixpoint Compilation_NS_compiler (d:MML_dB_exp) :  code :=
match d with
| Const_dB n ⇒ IConst n::nil
| ...
| Letm_dB d1 d2 ⇒ Compilation_NS_compiler d1 ++ILet::Compilation_NS_compiler d2 ++IEndLet::nil
| ...
| Lam_dB d ⇒ IClos (Compilation_NS_compiler d)::nil
| Mu_dB d ⇒ IClosr (Compilation_NS_compiler d)::nil
| App_dB d1 d2 ⇒ Compilation_NS_compiler d1 ++Compilation_NS_compiler d2 ++IApp::nil
end.
```

It is noteworthy how this time, instead of the function being an interpreter, it is a compiler, since it translates an expression instead of evaluating it. Also, it is not necessary to add a natural parameter to bound the recursion, given that the translation is decidable. This fact is guaranteed in Coq by using structural recursion (based on syntax) on the expression `d`.

The following lemma expresses that the compiler is sound regarding the natural semantics definition of the compilation:

```
Lemma Compilation_NS_compiler_soundness:
∀ d: MML_dB_exp,  ∀ c: code,
Compilation_NS d c →
Compilation_NS_compiler d = c.
```

Conversely, the next lemma expresses that the compiler is complete regarding the natural semantics definition of the compilation:

```
Lemma Compilation_NS_compiler_completeness:
∀ d: MML_dB_exp,  ∀ c: code,
Compilation_NS_compiler d = c →
Compilation_NS d c.
```

2.3.2. Correctness

To establish the correctness, we extend the compilation to values and environments once more, so after that we can formulate semantic preservation.

$$\frac{}{n \mapsdown n} \qquad \frac{}{b \mapsdown b} \qquad \frac{d \Downarrow c \qquad \Omega \downsquigarrow \Delta}{(\lambda.d)[\Omega] \mapsdown c[\Delta]} \qquad \frac{d \Downarrow c \qquad \Omega \downsquigarrow \Delta}{(\mu.\lambda.d)[\Omega] \mapsdown c[\Delta]_{rec}}$$

$$\frac{}{[\,] \downsquigarrow [\,]} \qquad \frac{v \mapsdown v_m \qquad \Omega \downsquigarrow \Delta}{v \cdot \Omega \downsquigarrow v_m \cdot \Delta}$$

To formulate correctness, we expect that if a nameless expression d is evaluated to a value v in an environment Ω; moreover, if c is the code resulting of the d compilation, and Δ is the resulting compilation of Ω; then, it must exist a machine value v_m that corresponds to the compilation of v and, when c is evaluated starting with the machine in a state (Δ, s), for any stack s, the evaluation takes the machine to the state $(\Delta, v_m \cdot s)$. Now, the correctness theorem is enunciated.

Theorem 2 (Correctness for termination)**.** *Let Ω be a nameless environment, Δ a machine environment, d a nameless expression, c a machine code, v a nameless value. If*

$$\Omega \vdash d \Rightarrow v, \qquad d \Downarrow c, \qquad \Omega \downsquigarrow \Delta$$

then there exists a machine value v_m such that $v \downzigzagarrow v_m$ and for all stack s,

$$\Delta, s \vdash c \Rightarrow (\Delta, v_m \cdot s)$$

Proof outline. We proceed by induction on the derivation of $\Omega \vdash d \Rightarrow v$. The base cases where d is a natural, a Boolean, a nameless variable, an abstraction, or a fixed point (recursive abstraction) are straightforward. In these cases, we exhibit a v_m such that $v \downzigzagarrow v_m$, since c is the result of d compilation, c is a single machine instruction hence $\Delta, s \vdash c \Rightarrow (\Delta, v_m \cdot s)$ follows simply by definition.

For the inductive cases where d is an arithmetic or comparison expression, a conditional, a local definition, or an application, the main idea is to use the induction hypothesis in tandem with Lemma 1. In such way the machine evaluation follows the structure of the $\Omega \vdash d \Rightarrow v$ derivation and the proof is simple and intuitive. □

The complete proof can be consulted in Appendix A. This theorem is written as follows in Coq:

```
Theorem Compilation_NS_correctness:
∀ O, ∀ d, ∀ v, MML_dB_NS O d v →
           ∀ c, Compilation_NS d c →
           ∀ D, Compilation_NS_env O D →
          ∃ mv, Compilation_NS_val v mv  ∧
           ∀ s, BSMSECD_NSC (D, s) c  (D, mv:: s).
```

We can immediately notice that due to the unifying use of natural semantics to define each of the compiler's components: source language, compilation and machine; the source language is mapped down, in a transparent way, to the target language (in this case, machine code) by means of the compilation. In this manner, to establish the correctness turns out to be easier, clearer, simpler and more intuitive than using an ad-hoc solution. For instance, in this case, it was unnecessary to previously define a closure of a relation, and it was also unnecessary to strengthen the hypothesis to prove the correctness theorem compared to the use of a function to define the compilation and small-step semantics in the machine.

3. Coinductive Natural Semantics

In this section, we will address the case in which the computations do not terminate, for which we will use coinductive natural semantics.

3.1. MiniML$^{\mathsf{dB}}$

In general, coinduction allows us to reason on infinite structures. In this way, taking into account the natural semantics design, we can employ a coinductive definition to express infinite evaluations of a language, in this case of MiniML$^{\mathsf{dB}}$. Following Leroy [14], we define the coinductive natural semantics for divergence (infinite evaluations) by the following predicate:

$$\Omega \vdash d \Rrightarrow$$

which can be read: in the Ω environment, the evaluation of the expression d diverges, is infinite, or, non-terminate.

That is, the infinite evaluations of MiniML$^{\mathsf{dB}}$ are defined by the coinductive interpretation of the following rules:

$$\frac{\Omega \vdash d_1 \Rrightarrow}{\Omega \vdash d_1 \star d_2 \Rrightarrow} \qquad\qquad \frac{\Omega \vdash d_1 \Rightarrow n_1 \qquad \Omega \vdash d_2 \Rrightarrow}{\Omega \vdash d_1 \star d_2 \Rrightarrow}$$

$$\frac{\Omega \vdash d_1 \overset{\infty}{\Rightarrow}}{\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \overset{\infty}{\Rightarrow}} \qquad \frac{\Omega \vdash d_1 \Rightarrow v_1 \qquad v_1 \cdot \Omega \vdash d_2 \overset{\infty}{\Rightarrow}}{\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \overset{\infty}{\Rightarrow}}$$

$$\frac{\Omega \vdash d_1 \overset{\infty}{\Rightarrow}}{\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \overset{\infty}{\Rightarrow}} \qquad \frac{\Omega \vdash d_1 \Rightarrow \mathit{true} \qquad \Omega \vdash d_2 \overset{\infty}{\Rightarrow}}{\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \overset{\infty}{\Rightarrow}} \qquad \frac{\Omega \vdash d_1 \Rightarrow \mathit{false} \qquad \Omega \vdash d_3 \overset{\infty}{\Rightarrow}}{\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \overset{\infty}{\Rightarrow}}$$

$$\frac{\Omega \vdash d_1 \overset{\infty}{\Rightarrow}}{\Omega \vdash d_1\ d_2 \overset{\infty}{\Rightarrow}} \qquad \frac{\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1] \qquad \Omega \vdash d_2 \overset{\infty}{\Rightarrow}}{\Omega \vdash d_1\ d_2 \overset{\infty}{\Rightarrow}} \qquad \frac{\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1] \qquad \Omega \vdash d_2 \overset{\infty}{\Rightarrow}}{\Omega \vdash d_1\ d_2 \overset{\infty}{\Rightarrow}}$$

$$\frac{\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1] \qquad \Omega \vdash d_2 \Rightarrow v_2 \qquad v_2 \cdot \Omega_1 \vdash d \overset{\infty}{\Rightarrow}}{\Omega \vdash d_1\ d_2 \overset{\infty}{\Rightarrow}}$$

$$\frac{\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1] \qquad \Omega \vdash d_2 \Rightarrow v_2 \qquad v_2 \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \overset{\infty}{\Rightarrow}}{\Omega \vdash d_1\ d_2 \overset{\infty}{\Rightarrow}}$$

Adopting the Leroy [14] convention, double horizontal lines denote coinductive interpretation, whereas single horizontal lines denote inductive interpretation.

The coinduction support in Coq is based on the work of Giménez [66]. In particular, Coq has native support of coinductive definitions. Likewise, a natural semantics definition can be encoded in Coq as an `Inductive` definition with type `Prop`, a coinductive natural semantics definition can be encoded in Coq as a `CoInductive` definition with type `Prop`. Hence, the MiniML$^{\mathsf{dB}}$ semantics for divergence can be written in Coq as follows:

```
CoInductive MML_dB_CNS: MML_dB_env → MML_dB_exp → Prop :=
| ...

| MML_dB_CNS_LetL:
      ∀ O, ∀ d1, MML_dB_CNS O d1 →
            ∀ d2, MML_dB_CNS O (Letm_dB d1 d2)

| MML_dB_CNS_LetR:
∀ O, ∀ d1, ∀ v1, MML_dB_NS O d1 v1 →
            ∀ d2, MML_dB_CNS (v1:: O) d2 →
                   MML_dB_CNS O (Letm_dB d1 d2)
| ...

| MML_dB_CNS_AppL:
      ∀ O, ∀ d1, MML_dB_CNS O d1 →
            ∀ d2, MML_dB_CNS O (App_dB d1 d2)

| MML_dB_CNS_AppR:
 ∀ O O1, ∀ d1 d, MML_dB_NS O d1 (Clos_dB d O1) →
            ∀ d2, MML_dB_CNS O d2 →
                   MML_dB_CNS O (App_dB d1 d2)

| MML_dB_CNS_AppRr:
 ∀ O O1, ∀ d1 d, MML_dB_NS O d1 (Closr_dB d O1) →
            ∀ d2, MML_dB_CNS O d2 →
                   MML_dB_CNS O (App_dB d1 d2)

| MML_dB_CNS_AppF:
 ∀ O O1, ∀ d1 d, MML_dB_NS O d1 (Clos_dB d O1) →
     ∀ d2, ∀ v2, MML_dB_NS O d2 v2 →
```

```
                  MML_dB_CNS (v2:: O1) d →
                  MML_dB_CNS O (App_dB d1 d2)

| MML_dB_CNS_AppFr:
 ∀ O O1, ∀ d1 d, MML_dB_NS O d1 (Closr_dB d O1) →
     ∀ d2, ∀ v2, MML_dB_NS O d2 v2 →
                  MML_dB_CNS (v2:: (Closr_dB d O1):: O1) d →
                  MML_dB_CNS O (App_dB d1 d2).
```

In the same manner as shown earlier in Section 2.1, where we verified that our interpreter `MML_dB_NS_interpreter` is sound regarding MiniML$^{\mathrm{dB}}$ natural semantics ; here, we must verify that it is sound regarding MiniML$^{\mathrm{dB}}$ coinductive natural semantics for non-termination, meaning, we must prove the next lemma:

```
Lemma MML_NS_interpreter_soundness_non_termination:
∀ O, ∀ d, MML_dB_CNS O d →
     ∀ n, MML_dB_NS_interpreter n O d = None.
```

This lemma states that if the evaluation of `d` does not terminate, whatever the `n` value of the fuel is, the interpreter will necessarily, eventually, run out of fuel (`None` means that the interpreter runs out of fuel).

Conversely, to verify that our interpreter is complete regarding MiniML$^{\mathrm{dB}}$ coinductive natural semantics, we must prove this lemma:

```
Lemma MML_NS_interpreter_completeness_non_termination: ∀ n, ∀ O, ∀ d,
MML_dB_NS_interpreter n O d = None →
not (∃ v, MML_dB_NS O d v) ∨
(∃ m, m > n ∧ ∃ v, MML_dB_NS_interpreter m O d = Some v).
```

This lemma says that if the interpreter runs out of fuel, then, there is not a finite evaluation of `d` or, in fact, it exists a finite evaluation of `d`, but more fuel is needed for the interpreter to be able to compute it. The proof of this lemma in Coq requires classic reasoning.

3.2. Modern SECD Machine

Let us now see how to express non-terminating computations in a machine. Leroy [14] uses small-step semantics to express infinitely many transitions in the MSECD. He defines the transition relation ‘ $\overset{\infty}{\Rightarrow}$ ’ coinductively as follows:

$$\frac{m_1 \to m_2 \qquad m_2 \overset{\infty}{\Rightarrow}}{m_1 \overset{\infty}{\Rightarrow}}$$

This relation can be written in Coq in the following manner:

```
CoInductive transinf: conf → Prop :=
| transinf_intro: ∀ m1 m2,
    MSECD_SS m1 m2 →
    transinf m2 →
    transinf m1.
```

However, by using this definition directly, it is not possible to prove the correctness of the compilation in the case of non-termination. The reason is that the Coq coinduction mechanism imposes the guard condition. The guard condition requires (at least) one rule (a constructor) of a coinductive definition to be used before the coinduction hypothesis is employed during a proof by coinduction. The solution offered by Leroy [14] is to define an auxiliary relation with which the proof can be carried out, and which is equivalent to the previous definition.

$$\dfrac{m \ \overset{\infty}{\underset{n}{\rightarrow}}}{m \ \overset{\infty}{\underset{n+1}{\rightarrow}}} \ (\overset{\infty}{\underset{n}{\rightarrow}}\text{-sleep}) \qquad\qquad \dfrac{m_1 \overset{+}{\rightarrow} m_2 \qquad m_2 \ \overset{\infty}{\underset{n'}{\rightarrow}}}{m_1 \ \overset{\infty}{\underset{n}{\rightarrow}}} \ (\overset{\infty}{\underset{n}{\rightarrow}}\text{-perform})$$

The most important property of the $\overset{\infty}{\underset{n}{\rightarrow}}$ relation, for our purposes, is that it allows the machine to remain in the same configuration, at most, a finite number n times ($\overset{\infty}{\underset{n}{\rightarrow}}$-sleep rule). This rule is crucial, as it is used for being able to comply with the guard condition when carrying out the correctness proof. At some point before n reaches 0, or necessarily when n arrives at 0, at least one transition ($\overset{\infty}{\underset{n}{\rightarrow}}$-perform rule) must be performed, in exchange for making a transition, the value of n is reset to any natural n', i.e., the possibility is given (again) to remain in the same configuration (this time, n' times at most). The relation $\overset{\infty}{\rightarrow}$ and the relation $\overset{\infty}{\underset{n}{\rightarrow}}$ are equivalent as stated in the following lemma:

Lemma 2. *Let m be a machine configuration, n any natural number,*

$$m \overset{\infty}{\rightarrow} \quad \textit{if and only if} \quad m \ \overset{\infty}{\underset{n}{\rightarrow}}$$

Proof outline. The *if* part is by coinduction. By $\overset{\infty}{\rightarrow}$ definition, necessarily $m \rightarrow m_1$ and $m_1 \overset{\infty}{\rightarrow}$, the result is obtained by applying the induction hypothesis on $m_1 \overset{\infty}{\rightarrow}$ and then by using the $\overset{\infty}{\underset{n}{\rightarrow}}$-perform rule. The *only if* part is also by coinduction. Assuming that if $m \ \overset{\infty}{\underset{n}{\rightarrow}}$ then $m \rightarrow m_1$ and $m_1 \ \overset{\infty}{\underset{n_1}{\rightarrow}}$, we apply the coinduction hypothesis on $m_1 \ \overset{\infty}{\underset{n_1}{\rightarrow}}$ and then the result follows by $\overset{\infty}{\rightarrow}$ definition. □

A more detailed proof can be consulted in Appendix A.

Compilation Correctness

To carry out the correctness proof, it is necessary to define a measure that indicates how many times the machine can remain in the same configuration based on the constructs of a language. The measure offered by Leroy (extended to cover all MiniML$^{\mathsf{dB}}$) is the following:

$$\begin{array}{rcl}
\|n\| = \|b\| = \|x\| = \|\lambda.d\| = \|\mu.\lambda.d\| & = & 0 \\
\|d_1 \star d_2\| & = & \|d_1\| + 1 \\
\|\texttt{let}\ d_1\ \texttt{in}\ d_2\| & = & \|d_1\| + 1 \\
\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\| & = & \|d_1\| + 1 \\
\|d_1\ d_2\| & = & \|d_1\| + 1
\end{array}$$

This is because it is possible that an evaluation step of a MiniML$^{\mathsf{dB}}$ expression d does not correspond, one to one in the same order, to a transition while evaluating $[\![d]\!]$ in the machine, causing the machine to stay at the same configuration, $\|d\|$ times, before performing a transition.

In this way, we are ready to state the correctness for the non-termination case, using the auxiliary relation $\overset{\infty}{\underset{n}{\rightarrow}}$ and strengthening the hypothesis, by the following lemma:

Lemma 3. *If* $\Omega \vdash d \overset{\infty}{\Rightarrow}$*, then* $([\![d]\!] \cdot c, [\![\Omega]\!], s) \ \overset{\infty}{\underset{\|d\|}{\rightarrow}}$ *for all codes c and stacks s.*

Proof outline. By coinduction. The main idea of the proof is to use Theorem 1 to evaluate the finite parts of d, to apply the coinduction hypothesis on the infinite parts of d, and to employ $\overset{\infty}{\underset{n}{\rightarrow}}$-sleep and $\overset{\infty}{\underset{n}{\rightarrow}}$-perform rules as convenient. □

A complete proof of this lemma can be consulted in Appendix A. This gives the possibility for formulating the correctness theorem directly with the $m \overset{\infty}{\rightarrow}$ relation, as Leroy [14] does, as follows:

Theorem 3. *If $\Omega \vdash d \overset{\infty}{\Rightarrow}$, then $([\![d]\!] \cdot c, [\![\Omega]\!], s) \overset{\infty}{\rightarrow}$ for all codes c and stacks s.*

Proof outline. The result is an immediate deduction from Lemma 3 followed by an application of Lemma 2. □

A step-by-step proof can be seen in Appendix A. This theorem is written in Coq as follows:

```
Theorem compile_evalinf:
∀ O, ∀ d, MML_dB_CNS O d →
∀ c, ∀ s, transinf (( compile d)++c,(compile_env O), s).
```

3.3. Big-Step MSECD Machine

This section introduces coinductive natural semantics to express non-terminating computations (infinite evaluations) in a machine. We illustrate its use with our big-step Modern SECD machine.

3.3.1. Rules of Non-terminating Computations

As with natural semantics (for finite evaluations, Section 2.3), coinductive natural semantics for divergence (infinite evaluations) is defined by the following two mutually dependent predicates:

$$\Delta, s \vdash i \overset{\infty}{\Rightarrow} \qquad\qquad \Delta, s \vdash c \overset{\infty}{\Rightarrow}$$

The first one reads: in the state (Δ, s) the instruction i diverges, whereas the second one reads: in the state (Δ, s) the code c diverges.

Thus, the infinite evaluations of the machine are defined by the coinductive interpretation of the following rules:

$$1)\ \frac{\Delta, s \vdash i \overset{\infty}{\Rightarrow}}{\Delta, s \vdash i \cdot c \overset{\infty}{\Rightarrow}} \qquad 2)\ \frac{\Delta, s \vdash i \Rightarrow (\Delta_1, s_1) \qquad \Delta_1, s_1 \vdash c \overset{\infty}{\Rightarrow}}{\Delta, s \vdash i \cdot c \overset{\infty}{\Rightarrow}}$$

$$3)\ \frac{\Delta, s \vdash c_1 \overset{\infty}{\Rightarrow}}{\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Rightarrow}} \qquad 4)\ \frac{\Delta, s \vdash c_2 \overset{\infty}{\Rightarrow}}{\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Rightarrow}}$$

$$5)\ \frac{v \cdot \Delta_1, s \vdash c \overset{\infty}{\Rightarrow}}{\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Rightarrow}} \qquad 6)\ \frac{v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \overset{\infty}{\Rightarrow}}{\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Rightarrow}}$$

The coinductive natural semantics of the machine is encoded in Coq in an analogous manner to the coinductive natural semantics of $\mathsf{MiniML}^{\mathsf{dB}}$ (it is just necessary to use a coinductive mutually dependent definition in Coq that parallels the mutually dependent predicates of the machine coinductive natural semantics). Likewise, the respective lemmas that express that the machine interpreter is sound and complete regarding the machine coinductive natural semantics are analogous to those of the $\mathsf{MiniML}^{\mathsf{dB}}$ interpreter. These formalizations can be seen in detail in [61].

We will give a brief explanation of the rules. When evaluating a code, it must begin by evaluating the first instruction i. The evaluation of i can be finite (rule 2) or infinite (rule 1). In the rule 1 case, if the first instruction diverges, then the complete code diverges. How can an instruction diverge?

Let us recall (Section 2.3) that due to the high level of abstraction of the big-step MSECD, in the case of termination, the ISel and IApp instructions are fully evaluated in a single big step, which includes the evaluation of their sub-codes and is why for these instructions it is necessary to define rules that allow expressing the possibility that their corresponding sub-codes diverge (rules 3–6). In the case of rule 2 , if the evaluation of the first instruction terminates but the remaining code diverges, then, the complete code (including the first instruction) diverges.

We can note here that in principle only rule 2 is necessary to express divergence in the machine since, intuitively, an instruction performs only a single basic operation and this rule is the analogous of the small-step semantics transition relation $\overset{\infty}{\rightarrow}$. However, as already mentioned, our big-step machine has two instructions, namely ISel and IApp, which are high-level (and therefore they evaluate different from their small-step semantics counterparts, performing not only single basic operations but a big-step sub-code evaluation). This is why these instructions require specific rules, while the remaining instructions, perform in fact only a single basic operation; for instance, $\mathsf{IConst}\ n$ push n on the stack. This is why these remaining instructions do not need specific rules.

In this way, the machine computations that do not terminate are completely defined. However, as in the case of the MSECD small-step semantics, we are facing, once more, the problem with Coq's guard condition. This means that similarly, we cannot prove correctness directly by using this relation. To solve this problem, we will present a variant of Leroy's solution adapted to coinductive natural semantics. This means that we must define an auxiliary relation equivalent to the previous relation, and which allows proving correctness. Below, we present the auxiliary relation:

$$\text{1)}\ \dfrac{\Delta, s \vdash i\ \overset{\infty}{\Mapsto}}{\Delta, s \vdash i \cdot c\ \underset{n}{\overset{\infty}{\Mapsto}}}$$

$$\text{3)}\ \dfrac{\Delta, s \vdash c_1\ \underset{n}{\overset{\infty}{\Mapsto}}}{\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2\ \overset{\infty}{\Mapsto}} \qquad \text{4)}\ \dfrac{\Delta, s \vdash c_2\ \underset{n}{\overset{\infty}{\Mapsto}}}{\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2\ \overset{\infty}{\Mapsto}}$$

$$\text{5)}\ \dfrac{v \cdot \Delta_1, s \vdash c\ \underset{n}{\overset{\infty}{\Mapsto}}}{\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp}\ \overset{\infty}{\Mapsto}} \qquad \text{6)}\ \dfrac{v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c\ \underset{n}{\overset{\infty}{\Mapsto}}}{\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp}\ \overset{\infty}{\Mapsto}}$$

$$\dfrac{\Delta, s \vdash c_1\ \underset{n}{\overset{\infty}{\Mapsto}}}{\Delta, s \vdash c_1 \cdot c_2\ \underset{n+1}{\overset{\infty}{\Mapsto}}}\ (\underset{n}{\overset{\infty}{\Mapsto}}\text{-sleep}) \qquad \dfrac{c_1 \neq [\,] \quad \Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1) \quad \Delta_1, s_1 \vdash c_2\ \underset{n'}{\overset{\infty}{\Mapsto}}}{\Delta, s \vdash c_1 \cdot c_2\ \underset{n}{\overset{\infty}{\Mapsto}}}\ (\underset{n}{\overset{\infty}{\Mapsto}}\text{-perform})$$

The rule ($\underset{n}{\overset{\infty}{\Mapsto}}$-sleep) is the improved analogous version of Leroy's rule ($\underset{n}{\overset{\infty}{\rightarrow}}$-sleep), since, in addition, it expresses that if in a code, the initial code c_1 diverges, then, no matter what the remaining code c_2 is, the code will diverge. The importance of this improvement is that it allows proving correctness without the need to strengthen the hypothesis. The rule ($\underset{n}{\overset{\infty}{\Mapsto}}$-perform) is analogous to the rule ($\underset{n}{\overset{\infty}{\rightarrow}}$-perform) of Leroy. We can note that rule 2 disappears (is no longer necessary) because it is subsumed by the ($\underset{n}{\overset{\infty}{\Mapsto}}$-perform) rule. For its part, the remaining rules (1 and 3–6) remain unchanged, i.e., they are analogous, but instead of the relation for code $\overset{\infty}{\Rightarrow}$ they use the relation $\underset{n}{\overset{\infty}{\Mapsto}}$ with any natural n.

It is worth remarking that the $\overset{\infty}{\Rightarrow}$ relation defines the machine computations that do not terminate in the expected way and it would be ideal working directly with it in Coq; however, because of the Coq guard condition it is not possible. This is why we have defined the $\overset{\infty}{\Mapsto}$ relation just to beat the Coq guard condition. Since we have defined the $\overset{\infty}{\Mapsto}$ relation (more precisely $\underset{n}{\overset{\infty}{\Mapsto}}$ relation for code and $\overset{\infty}{\Mapsto}$ relation for instructions) in a very similar way to the $\overset{\infty}{\Rightarrow}$ relation, we have used a very similar notation; nonetheless, we should be careful and notice the distinction between $\overset{\infty}{\Rightarrow}$ and $\overset{\infty}{\Mapsto}$.

The following lemma states that the two original mutually dependent relations are equivalent to the two auxiliary mutually dependent relations.

Lemma 4. *Let Δ be a machine environment, s a stack, i a machine instruction,*

$$\Delta, s \vdash i \overset{\infty}{\Rightarrow} \qquad \textit{if and only if} \qquad \Delta, s \vdash i \overset{\infty}{\Mapsto}$$

and, let c be a machine code, n any natural,

$$\Delta, s \vdash c \overset{\infty}{\Rightarrow} \qquad \textit{if and only if} \qquad \Delta, s \vdash c \underset{n}{\overset{\infty}{\Mapsto}}$$

Proof outline. The *if* part consist of: if $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$ then $\Delta, s \vdash i \overset{\infty}{\Mapsto}$, and if $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$ then $\Delta, s \vdash c \underset{n}{\overset{\infty}{\Mapsto}}$, i.e., of the following two cases:

1. If $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$ then $\Delta, s \vdash i \overset{\infty}{\Mapsto}$. Assuming 2, since the instructions' definition of $\overset{\infty}{\Rightarrow}$ and $\overset{\infty}{\Mapsto}$ are analogous, the proof proceeds simply by case analysis which are proved directly by $\overset{\infty}{\Mapsto}$ definition using 2 to obtain the required $\Delta, s \vdash c \underset{n}{\overset{\infty}{\Mapsto}}$ code premises.

2. If $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$ then $\Delta, s \vdash c \underset{n}{\overset{\infty}{\Mapsto}}$. By coinduction. The main idea is to use the coinduction hypothesis together with the $\underset{n}{\overset{\infty}{\Mapsto}}$-perform rule, and to apply 1 to obtain $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ premises when necessary.

The *only if* part consists of: If $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ then $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$, and if $\Delta, s \vdash c \underset{n}{\overset{\infty}{\Mapsto}}$ then $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$. That is, of the following two cases:

3. If $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ then $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$. Assuming 4, the proof is analogue to that of 1 going in the opposite direction (and using 4 instead of 2).

4. If $\Delta, s \vdash c \underset{n}{\overset{\infty}{\Mapsto}}$ then $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$. Assuming that, if $\Delta, s \vdash i \cdot c \underset{n}{\overset{\infty}{\Mapsto}}$ then $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ or there exists n_1, Δ_1, s_1, such that $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ and $\Delta_1, s_1 \vdash c \underset{n_1}{\overset{\infty}{\Mapsto}}$; The proof proceeds by coinduction. The key idea is to use the coinduction hypothesis together with $\underset{n}{\overset{\infty}{\Mapsto}}$ definition, in particular with the $\underset{n}{\overset{\infty}{\Mapsto}}$-perform rule, employing the assumption in the step which is required, and applying 3 to obtain $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$ premises when necessary. □

The details of this proof can be consulted in Appendix A.

3.3.2. Compilation Correctness

To prove correctness, we must use the auxiliary relation $\underset{n}{\overset{\infty}{\Mapsto}}$ along with a measure. Following an analogous argument to the case of the small-step MSECD, the measure $\|d\|$ also works here. In this way, the correctness for the non-termination case can be formulated by the following lemma:

Lemma 5 (Correctness for non-termination (auxiliary))**.** *Let Ω be a nameless environment, Δ a machine environment, d a nameless expression, c a machine code. If*

$$\Omega \vdash d \overset{\infty}{\Rightarrow}, \qquad d \Downarrow c, \qquad \Omega \wr \Delta$$

then, for all stack s,

$$\Delta, s \vdash c \underset{\|d\|}{\overset{\infty}{\Mapsto}}$$

Proof outline. We proceed by coinduction. The main idea is to mimic the $\mathsf{MiniML}^{\mathsf{dB}}$ d evaluation in the machine while evaluating c. To carry out this idea, for the finite parts of d (if any), we employ Theorem 2, whereas for the infinite parts of d, we apply the coinduction hypothesis. In addition, we use the $\underset{n}{\overset{\infty}{\Mapsto}}$ definition, including the $\underset{n}{\overset{\infty}{\Mapsto}}$-sleep and $\underset{n}{\overset{\infty}{\Mapsto}}$-perform rules, as necessary. □

A complete proof of this lemma appears in Appendix A. Finally, we enunciate the correctness theorem for the non-termination case of the machine, using the $\overset{\infty}{\Rightarrow}$ relation directly in the following manner:

Theorem 4 (Correctness for non-termination)**.** *Let Ω be a nameless environment, Δ a machine environment, d a nameless expression, c a machine code. If*

$$\Omega \vdash d \overset{\infty}{\Rightarrow}, \qquad d \Downarrow c, \qquad \Omega \wr \Delta$$

then, for all stack s,

$$\Delta, s \vdash c \overset{\infty}{\Rightarrow}$$

Proof outline. The result is a direct consequence of Lemma 5 followed by Lemma 4. □

A detailed step-by-step proof can be found in Appendix A. This theorem is written in Coq as follows:

```
Theorem Compilation_CNS_correctness:
∀ O, ∀ d, MML_dB_CNS O d →
     ∀ c, Compilation_NS d c →
     ∀ D, Compilation_NS_env O D →
     ∀ s, BSMSECD_CNSC (D, s) c.
```

4. Abstract to Coq Translation Algorithm

At this point, by means of our Mini-ML compiler, we have shown how from the (coinductive) natural semantics definition of each compiler component the corresponding formalization in Coq can be realized. It is our intention here to generalize this method and write it formally as an algorithm.

Algorithm 1 expresses how to translate a (coinductive) natural semantics definition of a compiler to its corresponding formalization in Coq.

We observe that the steps of the algorithm possess a high-level of abstraction. This is favorable in the sense that it provides freedom on how to actually implement them. We can even take advantage of this freedom by applying previous work. For instance, applying the results in [67], step 14 could be performed by generating a function from the `Inductive` definition I_N, corresponding to the natural semantics N.

Analyzing the algorithm, as we can note in step 35, the case in which the target language R of a translation T is a postfix representation requires a special treatment that merits explanation. Let V be the source language of T, d a construct of V, if c is the translation of d into R, then if R is a prefix representation when reasoning about the evaluation of c in Coq, necessarily a constructor s associated with the construct, must be used (at front) when starting the evaluation; in this way, the guard condition is fulfilled. Instead, if R is a postfix representation, then necessarily a constructor s associated with the construct, must be used but (behind), at the end of the evaluation; in this way, the Coq's guard condition is not fulfilled since it requires that it must be used at the start (at front). For instance, let `Plus e1 e2` be a construct in MiniML and let `Plus_dB d1 d2` be its translation in MiniML$^{\mathsf{dB}}$, then the evaluation of `Plus_dB d1 d2` will start using a `Plus_dB` constructor associated with the addition, at the start, and the Coq's guard condition will be satisfied. Instead, let `Plus_dB d1 d2` be a construct in MiniML$^{\mathsf{dB}}$ and `c1++c2++IAdd` its translation into code of the big-step MSECD, then the evaluation of `c1++c2++IAdd` will not start using an `IAdd` constructor associated with the addition, even when potentially, eventually, it will be used at the end in fact. Therefore, in this latter case, we must find a way to express and convince Coq's guard condition that the constructor will be in fact used, but at the end of the evaluation of c. This is exactly what the solution presented in Section 3.3 does, if the auxiliary relation is used, then initially the sleep rule constructor can be used, which allows the starting of the evaluation of c without using a constructor s associated with the construct, ensuring that potentially, eventually, such a s constructor will be in fact used (what is expressed in the constructor corresponding to the perform rule); the measure function $\|d\|$ indicates the number of constructors that will be used at the end of the evaluation of c. This is certainly a weakness of Coq's guard condition that turns out to be too inflexible in this case, and that is why we have had to resort to an indirect solution.

In fact, in our compiler, in the MiniML to $\mathsf{MiniML}^{\mathsf{dB}}$ translation it was not necessary to use this indirect solution at all because $\mathsf{MiniML}^{\mathsf{dB}}$ is a prefix representation. Instead, it was in fact necessary to use it in the $\mathsf{MiniML}^{\mathsf{dB}}$ to big-step MSECD machine code generation because machine code is a postfix representation.

Algorithm 1: Translation of a Compiler Definition to Coq (first part)

Input: A (coinductive) natural semantics definition of a total correct compiler (where all-natural semantics and translations are deterministic)

Output: A Coq formalization of the compiler (from which a verified implementation can be obtained)

```
1  foreach language L do
2      Let $A$ be the abstract syntax of $L$;
3      Emit an Inductive definition $I_A$ with type Set;
4      foreach language construct $c \in A$ do
5          Add the constructor $s$, corresponding to the construct $c$, to $I_A$;
6      end
7      Emit an Extraction command with argument the $I_A$ definition;
8      Let $N$ be the natural semantics of $L$;
9      Emit an Inductive definition $I_N$ with type Prop;
10     foreach rule $r \in N$ do
11         Add the constructor $s$, corresponding to the rule $r$, to $I_N$;
12     end
13     Emit a Lemma that states $N$ determinism;
14     Emit a Fixpoint function (interpreter) $i$ that mimics the $N$ natural semantics;
15     Emit a Lemma enunciating that the interpreter $i$ is sound regarding $N$ natural semantics;
16     Emit a Lemma enunciating that the interpreter $i$ is complete regarding $N$ natural semantics;
17     Let $CoN$ be the coinductive natural semantics of $L$;
18     Emit a CoInductive definition $C_{CoN}$ with type Prop;
19     foreach rule $r \in CoN$ do
20         Add the constructor $s$, corresponding to the rule $r$, to $C_{CoN}$;
21     end
22     Emit a Lemma enunciating that the interpreter $i$ is sound regarding $CoN$ coinductive natural
         semantics;
23     Emit a Lemma enunciating that the interpreter $i$ is complete regarding $CoN$ coinductive
         natural semantics;
24     Emit an Extraction command with the interpreter $i$ as argument;
25 end
26 ...
```

Algorithm 1: Translation of a Compiler Definition to Coq (second part)

```
foreach translation $T$ do
    Emit an Inductive definition $I_T$ with type Prop;
    foreach translation rule $r \in T$ do
        Add the constructor $s$, corresponding to rule $r$, to $I_T$;
    end
    Emit a Lemma that states $T$ determinism;
    Emit a Fixpoint function (compiler) $c$ that mimics the $T$ translation;
    Emit a Theorem enunciating the translation $T$ correctness for termination;
    Let $R$ be the target language of the translation $T$;
    if $R$ is a postfix representation then
        Emit an auxiliary CoInductive definition $C'_{CoN}$ analogous to $C_{CoN}$, but including a
          natural $n$ as additional term;
        Add the constructor $s$, corresponding to the adapted improved sleep rule, to $C'_{CoN}$;
        Add the constructor $p$, corresponding to the adapted perform rule, to $C'_{CoN}$;
        foreach rule $r \in CoN$ do
            if $r$ is subsumed by the improved sleep rule or the perform rule then
                Remove the constructor $s$, corresponding to the rule $r$, from $C'_{CoN}$;
            end
        end
        Emit a Lemma that states the equivalence between $C_{CoN}$ and $C'_{CoN}$;
        Let $V$ be the source language of $T$;
        Define a size function $\|d\|$ over the constructs of $V$ where
        if $d$ is an atom then
            $\|d\| = 0$;
        else if $d$ is composed by $d_1, \ldots, d_n$ then
            $\|d\| = 1 + \|d_1\|$;
        end
        Emit a Fixpoint function (left height) $h$ that mimics the $\|d\|$ size;
        Emit a Lemma enunciating the translation $T$ correctness for non-termination
          (using $C'_{CoN}$ and $h$);
    end
    Emit a Theorem enunciating the translation $T$ correctness for non-termination (using $C_{CoN}$);
    Emit an Extraction command with the $c$ compiler as argument;
end
```

5. Conclusions

Natural semantics is a simple, easy, and intuitive formalism widely known and used in the literature to define the semantics of programming languages.

In this work, we extended (coinductive) natural semantics to present it as a unifying framework for the verification of total correctness of compilers in Coq. This way, we present a solution to the problem of having a simple, easy, clear, and intuitive framework to perform this task in this proof assistant. By means of this framework, one can obtain standalone executable verified compiler.

Although we have not illustrated it here, natural semantics can also be used to express and verify the static semantics of a language. For instance, in [68] the Mini-ML static semantics is verified (although it is not possible to obtain a verified semantic analyzer). In future work, we plan to extend this use of natural semantics to make it possible to obtain a verified semantic analyzer.

To have a full compiler framework, we must address lexing and parsing too. So, we envisioned that natural 'semantics' can also be used to perform these tasks. This inspiration comes from the observation that, as stated by Kahn [11], natural deduction is at the heart of natural semantics, so we are looking for a natural deduction-based parsing strategy, fortunately, it already exists a parsing technique with these features since long time ago. In the logic programming community, parsing as deduction [69,70] is a well-known and established natural deduction-based parsing framework. Therefore, since both: natural semantics and parsing as deduction are based on natural deduction, we believe that we can abstract both in a single formalism able to express both: syntax and semantics. In this way, it would achieve natural 'semantics' as full compiler verification framework in Coq.

Author Contributions: Conceptualization, A.Z.; methodology, A.Z.; software, A.Z.; validation, A.Z.; formal analysis, A.Z.; investigation, A.Z.; resources, G.B.-E.; writing—original draft preparation, A.Z.; writing—review and editing, A.Z. and G.B.-E.; visualization, A.Z.; supervision, G.B.-E.; project administration, G.B.-E.; funding acquisition, G.B.-E. Both authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by CONACYT, grant number A1-S-27780, and DGAPA UNAM, grant number PAPIIT-A401219.

Acknowledgments: The first author sincerely thanks Xavier Leroy for inspiring him to work with coinductive natural semantics in Coq. In particular, for revising his preliminary results regarding the equivalence of $\overset{co}{\Rightarrow}$ with the union of $\Rightarrow$ and $\overset{\infty}{\Rightarrow}$. Finally, for encouraging him to write his results and extend his work. The first author is grateful to David de Frutos-Escrig for providing advice during this work. Special thanks to Veronica Dahl who offered guidance, and useful insight into Prolog parsing techniques. Our deepest gratitude goes to our anonymous reviewers whose accurate, appropriate and constructive comments have led to a valuable improvement of this work.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Proofs

Theorem A1. *If* $\Omega \vdash d \Rightarrow v$*, then* $([\![d]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow{+} (c, [\![\Omega]\!], [\![v]\!] \cdot s)$ *for all codes* c *and stacks* s*.*

Proof. By induction on $\Omega \vdash d \Rightarrow v$.
Base cases:

(i) $d = n$. By hypothesis $\Omega \vdash n \Rightarrow n$. Since by definition $[\![n]\!] = \mathsf{IConst}\ n$ and $[\![n]\!] = n$ we must prove $(\mathsf{IConst}\ n \cdot c, [\![\Omega]\!], s) \xrightarrow{+} (c, [\![\Omega]\!], n \cdot s)$, which follows simply by definition of the machine small-step semantics transition $\rightarrow$ corresponding to the $\mathsf{IConst}\ n$ instruction.

(ii) $d = b$. Analogous to case i. Hypothesis $\Omega \vdash b \Rightarrow b$; $[\![b]\!] = \mathsf{IConstb}\ b$ and $[\![b]\!] = b$.

(iii) $d = \underline{i}$. The proof follows from the fact that if $\Omega \vdash \underline{i} \longmapsto v$ then $[\![\Omega]\!](i) = [\![v]\!]$ which is proved by straightforward induction on Ω.

(iv) $d = \lambda.d$. Analogous to case i. Hypothesis $\Omega \vdash \lambda.d \Rightarrow (\lambda.d)[\Omega]$; $[\![\lambda.d]\!] = \mathsf{IClos}([\![d]\!] \cdot \mathsf{IRet})$ and $[\![(\lambda.d)[\Omega]]\!] = ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega]\!]]$.

(v) $d = \mu.\lambda.d$. Analogous to case i. Hypothesis $\Omega \vdash \mu.\lambda.d \Rightarrow (\mu.\lambda.d)[\Omega]$; $[\![\mu.\lambda.d]\!] = \mathsf{IClos}_{rec}([\![d]\!] \cdot \mathsf{IRet})$ and $[\![(\mu.\lambda.d)[\Omega]]\!] = ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega]\!]]_{rec}$.

Inductive cases:

(vi) $d = d_1 \star d_2$.

By hypothesis $\Omega \vdash d_1 \Rightarrow n_1$, $\Omega \vdash d_2 \Rightarrow n_2$, $\Omega \vdash d_1 \star d_2 \Rightarrow n_1 \star n_2$. We must prove $([\![d_1 \star d_2]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow{+} (c, [\![\Omega]\!], [\![n_1 \star n_2]\!] \cdot s)$.

(a) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IOp} \cdot c, [\![\Omega]\!], s) \xrightarrow{+} (c, [\![\Omega]\!], n_1 \star n_2 \cdot s)$ by definition $[\![d_1 \star d_2]\!] = [\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IOp}$ and $[\![n_1 \star n_2]\!] = n_1 \star n_2$.

(b) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IOp} \cdot c, [\![\Omega]\!], s) \xrightarrow{+} ([\![d_2]\!] \cdot \mathsf{IOp} \cdot c, [\![\Omega]\!], n_1 \cdot s)$ by $\xrightarrow{+}$ transitivity, applying induction hypothesis, and by definition $[\![n_1]\!] = n_1$.

(c) $(\llbracket d_2 \rrbracket \cdot \mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, n_1 \cdot s) \overset{+}{\rightarrow} (\mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, n_2 \cdot n_1 \cdot s)$ by $\overset{+}{\rightarrow}$ transitivity, applying induction hypothesis, and by definition $\llbracket n_2 \rrbracket = n_2$.

(d) $(\mathsf{IOp} \cdot c, \llbracket \Omega \rrbracket, n_2 \cdot n_1 \cdot s) \overset{+}{\rightarrow} (c, \llbracket \Omega \rrbracket, n_1 \star n_2 \cdot s)$ by $\rightarrow$ definition of IOp.

(vii) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

By hypothesis $\Omega \vdash d_1 \Rightarrow true, \quad \Omega \vdash d_2 \Rightarrow v$. We must prove
$(\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\rightarrow} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$.

(a) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\rightarrow} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$
by definition $\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin})$.

(b) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\rightarrow}$
$(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, true \cdot s)$ by $\overset{+}{\rightarrow}$ transitivity, and applying induction hypothesis

(c) $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, true \cdot s) \overset{+}{\rightarrow} (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, [\,]) \cdot s)$ by $\rightarrow$ definition of ISel.

(d) $(\llbracket d_2 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, [\,]) \cdot s) \overset{+}{\rightarrow} (\mathsf{IJoin}, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot (c, [\,]) \cdot s)$ by $\overset{+}{\rightarrow}$ transitivity, and applying induction hypothesis.

(e) $(\mathsf{IJoin}, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot (c, [\,]) \cdot s) \overset{+}{\rightarrow} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$ by $\rightarrow$ definition of IJoin.

(viii) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

By hypothesis $\Omega \vdash d_1 \Rightarrow false, \quad \Omega \vdash d_3 \Rightarrow v$. We must prove
$(\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\rightarrow} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$.

(a) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\rightarrow} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$
by definition $\llbracket \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin})$.

(b) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\rightarrow}$
$(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, false \cdot s)$ by $\overset{+}{\rightarrow}$ transitivity, and applying induction hypothesis

(c) $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, false \cdot s) \overset{+}{\rightarrow} (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, [\,]) \cdot s)$ by $\rightarrow$ definition of ISel.

(d) $(\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, [\,]) \cdot s) \overset{+}{\rightarrow} (\mathsf{IJoin}, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot (c, [\,]) \cdot s)$ by $\overset{+}{\rightarrow}$ transitivity, and applying induction hypothesis.

(e) $(\mathsf{IJoin}, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot (c, [\,]) \cdot s) \overset{+}{\rightarrow} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$ by $\rightarrow$ definition of IJoin.

(ix) $d = \texttt{let}\ d_1\ \texttt{in}\ d_2$.

By hypothesis $\Omega \vdash d_1 \Rightarrow v_1, \quad v_1 \cdot \Omega \vdash d_2 \Rightarrow v, \quad \Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \Rightarrow v$. We must prove

$(\llbracket \texttt{let}\ d_1\ \texttt{in}\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\rightarrow} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$.

(a) $(\llbracket d_1 \rrbracket \cdot \mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet} \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\rightarrow} (c, \llbracket \Omega \rrbracket, \llbracket v \rrbracket \cdot s)$
by definition $\llbracket \texttt{let}\ d_1\ \texttt{in}\ d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ILet} \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IEndLet}$.

(b) $([\![d_1]\!] \cdot \mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c,\ [\![\Omega]\!], s) \xrightarrow{+} (\mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c, [\![\Omega]\!], [\![v_1]\!] \cdot s)$ by $\xrightarrow{+}$ transitivity, and applying induction hypothesis.

(c) $(\mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c,\ [\![\Omega]\!], [\![v_1]\!] \cdot s) \xrightarrow{+} ([\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c, [\![v_1]\!] \cdot [\![\Omega]\!], s)$ by $\rightarrow$ definition of ILet.

(d) $([\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c,\ [\![v_1]\!] \cdot [\![\Omega]\!], s) \xrightarrow{+} (\mathsf{IEndLet} \cdot c, [\![v_1]\!] \cdot [\![\Omega]\!], [\![v]\!] \cdot s)$ by $\xrightarrow{+}$ transitivity, and applying induction hypothesis.

(e) $(\mathsf{IEndLet} \cdot c, [\![v_1]\!] \cdot [\![\Omega]\!], [\![v]\!] \cdot s) \xrightarrow{+} (c, [\![\Omega]\!], [\![v]\!] \cdot s)$ by $\rightarrow$ definition of IEndLet.

(x) $d = d_1\ d_2$.

By hypothesis $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$, $\Omega \vdash d_2 \Rightarrow v_2$, $v_2 \cdot \Omega_1 \vdash d \Rightarrow v$, $\Omega \vdash d_1\ d_2 \Rightarrow v$. We must prove $([\![d_1\ d_2]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow{+} (c, [\![\Omega]\!], [\![v]\!] \cdot s)$.

(a) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], s) \xrightarrow{+} (c, [\![\Omega]\!], [\![v]\!] \cdot s)$ by definition $[\![d_1\ d_2]\!] = [\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp}$

(b) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], s) \xrightarrow{+} ([\![d_2]\!] \cdot \mathsf{IApp}\cdot, [\![\Omega]\!], [\![(\lambda.d)[\Omega_1]]\!] \cdot s)$ by $\xrightarrow{+}$ transitivity, and applying induction hypothesis.

(c) $([\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], [\![(\lambda.d)[\Omega_1]]\!] \cdot s) \xrightarrow{+} (\mathsf{IApp} \cdot c, [\![\Omega]\!], [\![v_2]\!] \cdot [\![(\lambda.d)[\Omega_1]]\!] \cdot s)$ by $\xrightarrow{+}$ transitivity, and applying induction hypothesis.

(d) $([\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], [\![(\lambda.d)[\Omega_1]]\!] \cdot s) \xrightarrow{+} (\mathsf{IApp} \cdot c, [\![\Omega]\!], [\![v_2]\!] \cdot ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]] \cdot s)$ by definition $[\![(\lambda.d)[\Omega_1]]\!] = ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]]$.

(e) $(\mathsf{IApp} \cdot c, [\![\Omega]\!], [\![v_2]\!] \cdot ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]] \cdot s) \xrightarrow{+} ([\![d]\!] \cdot \mathsf{IRet}, [\![v_2]\!] \cdot [\![\Omega_1]\!], (c, [\![\Omega]\!]) \cdot s)$ by $\rightarrow$ definition of IApp.

(f) $([\![d]\!] \cdot \mathsf{IRet}, [\![v_2]\!] \cdot [\![\Omega_1]\!], (c, [\![\Omega]\!]) \cdot s) \xrightarrow{+} (\mathsf{IRet}, [\![v_2]\!] \cdot [\![\Omega_1]\!], [\![v]\!] \cdot (c, [\![\Omega]\!]) \cdot s)$ by $\xrightarrow{+}$ transitivity and applying induction hypothesis.

(g) $(\mathsf{IRet}, [\![v_2]\!] \cdot [\![\Omega_1]\!], [\![v]\!] \cdot (c, [\![\Omega]\!]) \cdot s) \xrightarrow{+} (c, [\![\Omega]\!], [\![v]\!] \cdot s)$ by $\rightarrow$ definition of IRet.

(xi) $d = d_1\ d_2$.

By hypothesis $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$, $\Omega \vdash d_2 \Rightarrow v_2$, $v_2 \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \Rightarrow v$,
$\Omega \vdash d_1\ d_2 \Rightarrow v$. We must prove $([\![d_1\ d_2]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow{+} (c, [\![\Omega]\!], [\![v]\!] \cdot s)$.

(a) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], s) \xrightarrow{+} (c, [\![\Omega]\!], [\![v]\!] \cdot s)$ by definition $[\![d_1\ d_2]\!] = [\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp}$

(b) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], s) \xrightarrow{+} ([\![d_2]\!] \cdot \mathsf{IApp}\cdot, [\![\Omega]\!], [\![(\mu.\lambda.d)[\Omega_1]]\!] \cdot s)$ by $\xrightarrow{+}$ transitivity, and applying induction hypothesis.

(c) $([\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], [\![(\mu.\lambda.d)[\Omega_1]]\!] \cdot s) \xrightarrow{+} (\mathsf{IApp} \cdot c, [\![\Omega]\!], [\![v_2]\!] \cdot [\![(\mu.\lambda.d)[\Omega_1]]\!] \cdot s)$ by $\xrightarrow{+}$ transitivity, and applying induction hypothesis.

(d) $([\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], [\![(\mu.\lambda.d)[\Omega_1]]\!] \cdot s) \xrightarrow{+} (\mathsf{IApp} \cdot c, [\![\Omega]\!], [\![v_2]\!] \cdot ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]]_{rec} \cdot s)$ by definition $[\![(\mu.\lambda.d)[\Omega_1]]\!] = ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]]_{rec}$.

(e) $(\mathsf{IApp} \cdot c, [\![\Omega]\!], [\![v_2]\!] \cdot ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]]_{rec} \cdot s) \xrightarrow{+}$
$([\![d]\!] \cdot \mathsf{IRet}, [\![v_2]\!] \cdot ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]]_{rec} \cdot [\![\Omega_1]\!], (c, [\![\Omega]\!]) \cdot s)$ by $\rightarrow$ definition of IApp.

(f) $([\![d]\!] \cdot \mathsf{IRet}, [\![v_2]\!] \cdot ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]]_{rec} \cdot [\![\Omega_1]\!], (c, [\![\Omega]\!]) \cdot s) \xrightarrow{+}$
$(\mathsf{IRet}, [\![v_2]\!] \cdot ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]]_{rec} \cdot [\![\Omega_1]\!], [\![v]\!] \cdot (c, [\![\Omega]\!]) \cdot s)$ by $\xrightarrow{+}$ transitivity and applying induction hypothesis.

(g) $(\mathsf{IRet}, [\![v_2]\!] \cdot ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]]_{rec} \cdot [\![\Omega_1]\!], [\![v]\!] \cdot (c, [\![\Omega]\!]) \cdot s) \xrightarrow{+} (c, [\![\Omega]\!], [\![v]\!] \cdot s)$ by $\rightarrow$ definition of IRet. □

Lemma A1. *Let Δ, Δ_1, Δ_2 be machine environments; s, s_1, s_2 stacks; c_1, c_2 machine codes. If*

$$\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1), \qquad \Delta_1, s_1 \vdash c_2 \Rightarrow (\Delta_2, s_2)$$

then

$$\Delta, s \vdash c_1 \cdot c_2 \Rightarrow (\Delta_2, s_2)$$

Proof. By induction on $\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1)$.
Base case:

(i) $c_1 = []$.

Hypothesis $\Delta, s \vdash [] \Rightarrow (\Delta, s)$, $\Delta, s \vdash c_2 \Rightarrow (\Delta_2, s_2)$. We must prove $\Delta, s \vdash [] \cdot c_2 \Rightarrow (\Delta_2, s_2)$, since $[] \cdot c_2 = c_2$ our goal reduces to $\Delta, s \vdash c_2 \Rightarrow (\Delta_2, s_2)$ which follows by hypothesis.

Inductive case:

(ii) $c_1 \neq []$ that is $c_1 = i \cdot c$.

Hypothesis $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$, $\Delta_1, s_1 \vdash c \Rightarrow (\Delta_2, s_2)$, $\Delta_2, s_2 \vdash c_2 \Rightarrow (\Delta_3, s_3)$. We must prove $\Delta, s \vdash i \cdot c \cdot c_2 \Rightarrow (\Delta_3, s_3)$.

(a) Applying induction hypothesis on $\Delta_1, s_1 \vdash c \Rightarrow (\Delta_2, s_2)$ and $\Delta_2, s_2 \vdash c_2 \Rightarrow (\Delta_3, s_3)$ we have $\Delta_1, s_1 \vdash c \cdot c_2 \Rightarrow (\Delta_3, s_3)$.

(b) Using $(\Delta, s) \vdash i \Rightarrow (\Delta_1, s_1)$ and $\Delta_1, s_1 \vdash c \cdot c_2 \Rightarrow (\Delta_3, s_3)$ by $\Rightarrow$ definition we conclude $\Delta, s \vdash i \cdot c \cdot c_2 \Rightarrow (\Delta_3, s_3)$. □

Theorem A2 (Correctness for termination). *Let Ω be a nameless environment, Δ a machine environment, d a nameless expression, c a machine code, v a nameless value. If*

$$\Omega \vdash d \Rightarrow v, \quad d \Downarrow c, \quad \Omega \wr \Delta$$

then, there exists a machine value v_m such that $v \updownarrow v_m$ and for all stack s,

$$\Delta, s \vdash c \Rightarrow (\Delta, v_m \cdot s)$$

Proof. By induction on $\Omega \vdash d \Rightarrow v$.

Base cases:

(i) $d = n$.

Hypothesis $\Omega \vdash n \Rightarrow n$, $n \Downarrow c$, $\Omega \wr \Delta$. We claim that there exists $v_m = n$, the proof of $n \updownarrow n$ follows by definition. Using the $n \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $c = \mathsf{IConst}\ n$. We are now in a position to prove the main goal $\Delta, s \vdash \mathsf{IConst}\ n \Rightarrow (\Delta, n \cdot s)$ which follows by $\Rightarrow$ definition of IConst.

(ii) $d = b$. Analogous to case i.

Hypothesis $\Omega \vdash b \Rightarrow b, \quad b \Downarrow c, \quad \Omega \wr \Delta$. We claim that there exists $v_m = b$, the proof of $b ↧ b$ follows by definition. Using the $b \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $c = \mathsf{IConstb}\ b$. We are now in a position to prove the main goal $\Delta, s \vdash \mathsf{IConstb}\ b \Rightarrow (\Delta, b \cdot s)$ which follows by $\Rightarrow$ definition of $\mathsf{IConstb}$.

(iii) $d = \underline{i}$.

The proof relies on the fact that if $\Omega \vdash \underline{i} \longmapsto v, \quad \Omega \wr \Delta, \quad v ↧ v_m$ then $\Delta \vdash \underline{i} \longmapsto v_m$ which is proved by straightforward induction on $\Omega \vdash \underline{i} \longmapsto v$. (Also, it can by proved by induction on Ω.)

(iv) $d = \lambda.d$. Analogous to case i.

Hypothesis $\Omega \vdash \lambda.d \Rightarrow (\lambda.d)[\Omega], \quad \lambda.d \Downarrow c, \quad \Omega \wr \Delta$. Using the $\lambda.d \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d \Downarrow c_1, \quad c = \mathsf{IClos}\ c_1$. We claim that there exists $v_m = c_1[\Delta]$, the proof of $(\lambda.d)[\Omega] ↧ c_1[\Delta]$ follows by definition, $d \Downarrow c_1$ and $\Omega \wr \Delta$. We are now in a position to prove the main goal $\Delta, s \vdash \mathsf{IClos}\ c_1 \Rightarrow (\Delta, c_1[\Delta] \cdot s)$ which follows by $\Rightarrow$ definition of IClos.

(v) $d = \mu.\lambda.d$. Analogous to case i.

Hypothesis $\Omega \vdash \mu.\lambda.d \Rightarrow (\mu.\lambda.d)[\Omega], \quad \mu.\lambda.d \Downarrow c, \quad \Omega \wr \Delta$. Using the $\mu.\lambda.d \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d \Downarrow c_1, \quad c = \mathsf{IClos}_{rec}\ c_1$. We claim that there exists $v_m = c_1[\Delta]_{rec}$, the proof of $(\mu.\lambda.d)[\Omega] ↧ c_1[\Delta]_{rec}$ follows by definition, $d \Downarrow c_1$ and $\Omega \wr \Delta$. We are now in a position to prove the main goal $\Delta, s \vdash \mathsf{IClos}_{rec}\ c_1 \Rightarrow (\Delta, c_1[\Delta]_{rec} \cdot s)$ which follows by $\Rightarrow$ definition of IClos_{rec}.

Inductive cases:

(vi) $d = d_1 \star d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow n_1, \quad \Omega \vdash d_2 \Rightarrow n_2, \quad \Omega \vdash d_1 \star d_2 \Rightarrow n_1 \star n_2, \quad d_1 \star d_2 \Downarrow c, \quad \Omega \wr \Delta$.

We claim that there exists $v_m = n_1 \star n_2$, the proof of $n_1 \star n_2 ↧ n_1 \star n_2$ follows by definition. Using the $d_1 \star d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad c = c_1 \cdot c_2 \cdot \mathsf{IOp}$. We are now in position to prove the main goal $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IOp} \Rightarrow (\Delta, n_1 \star n_2 \cdot s)$.

(a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, n_1 \cdot s)$ by induction hypothesis.

(b) $\Delta, n_1 \cdot s \vdash c_2 \Rightarrow (\Delta, n_2 \cdot n_1 \cdot s)$ by induction hypothesis.

(c) $\Delta, s \vdash c_1 \cdot c_2 \Rightarrow (\Delta, n_2 \cdot n_1 \cdot s)$ by Lemma A1 on vi.a and vi.b.

(d) $\Delta, n_2 \cdot n_1 \cdot s \vdash \mathsf{IOp} \Rightarrow (\Delta, n_1 \star n_2 \cdot s)$ by definition.

(e) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IOp} \Rightarrow (\Delta, n_1 \star n_2 \cdot s)$ by Lemma A1 on vi.c and vi.d.

(vii) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

Hypothesis $\Omega \vdash d_1 \Rightarrow true, \quad \Omega \vdash d_2 \Rightarrow v, \quad \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c, \quad \Omega \wr \Delta$. Using the $\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad d_3 \Downarrow c_3$, $c = c_1 \cdot \mathsf{ISel}\ c_2\ c_3$. We are now in position to prove the main goal $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \Rightarrow (\Delta, v_m \cdot s)$.

(a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, true \cdot s)$ by induction hypothesis.

(b) $\Delta, s \vdash c_2 \Rightarrow (\Delta, v_m \cdot s)$ by induction hypothesis (such that $v ↧ v_m$).

(c) $\Delta, true \cdot s \vdash \mathsf{ISel}\ c_2\ c_3 \Rightarrow (\Delta, v_m \cdot s)$ by definition using viii.b.

(d) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \Rightarrow (\Delta, v_m \cdot s)$ by Lemma A1 on viii.a and viii.c.

(viii) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

Hypothesis $\Omega \vdash d_1 \Rightarrow false$, $\Omega \vdash d_3 \Rightarrow v$, $\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c$, $\Omega \wr \Delta$. Using the $\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $d_3 \Downarrow c_3$, $c = c_1 \cdot \mathsf{ISel}\ c_2\ c_3$. We are now in position to prove the main goal $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \Rightarrow (\Delta, v_m \cdot s)$.

(a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, false \cdot s)$ by induction hypothesis.

(b) $\Delta, s \vdash c_3 \Rightarrow (\Delta, v_m \cdot s)$ by induction hypothesis (such that $v \wr v_m$).

(c) $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_2\ c_3 \Rightarrow (\Delta, v_m \cdot s)$ by definition using viii.b.

(d) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \Rightarrow (\Delta, v_m \cdot s)$ by Lemma A1 on viii.a and viii.c.

(ix) $d = \texttt{let}\ d_1\ \texttt{in}\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow v_1$, $v_1 \cdot \Omega \vdash d_2 \Rightarrow v$, $\Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \Rightarrow v$, $\texttt{let}\ d_1\ \texttt{in}\ d_2 \Downarrow c$, $\Omega \wr \Delta$. Using the $\texttt{let}\ d_1\ \texttt{in}\ d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet}$. We are now in position to prove the main goal
$\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \Rightarrow (\Delta, v_m \cdot s)$

(a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, v_{m_1} \cdot s)$ by induction hypothesis (such that $v_1 \wr v_{m_1}$).

(b) $\Delta, v_{m_1} \cdot s \vdash \mathsf{ILet} \Rightarrow (v_{m_1} \cdot \Delta, s)$ by definition.

(c) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \Rightarrow (v_{m_1} \cdot \Delta, s)$ by Lemma A1 on ix.a and ix.b.

(d) $v_{m_1} \cdot \Delta, s \vdash c_2 \Rightarrow (v_{m_1} \cdot \Delta, v_m \cdot s)$ by induction hypothesis (such that $v \wr v_m$).

(e) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \Rightarrow (v_{m_1} \cdot \Delta, v_m \cdot s)$ by Lemma A1 on ix.c and ix.d.

(f) $v_{m_1} \cdot \Delta, v_m \cdot s \vdash \mathsf{IEndLet} \Rightarrow (\Delta, v_m \cdot s)$ by definition.

(g) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \Rightarrow (\Delta, v_m \cdot s)$ by Lemma A1 on ix.e and ix.f.

(x) $d = d_1\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$, $\Omega \vdash d_2 \Rightarrow v_2$, $v_2 \cdot \Omega_1 \vdash d \Rightarrow v$, $\Omega \vdash d_1\ d_2 \Rightarrow v$, $d_1\ d_2 \Downarrow c$, $\Omega \wr \Delta$. Using the $d_1\ d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot c_2 \cdot \mathsf{IApp}$.

We are now in position to prove the main goal $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Rightarrow (\Delta, m_v \cdot s)$.

(a) $d \Downarrow c$, $\Omega_1 \wr \Delta_1$, $\Delta, s \vdash c_1 \Rightarrow (\Delta, c[\Delta_1] \cdot s)$ by induction hypothesis (such that $(\lambda.d)[\Omega_1] \wr c[\Delta_1]$).

(b) $\Delta, c[\Delta_1] \cdot s \vdash c_2 \Rightarrow (\Delta, v_{m_2} \cdot c[\Delta_1] \cdot s)$ by induction hypothesis (such that $v_2 \wr v_{m_2}$).

(c) $\Delta, s \vdash c_1 \cdot c_2 \Rightarrow (\Delta, v_{m_2} \cdot c[\Delta_1] \cdot s)$ by Lemma A1 on x.a and x.b.

(d) $v_{m_2} \cdot \Delta_1, s \vdash c \Rightarrow (v_{m_2} \cdot \Delta_1, v_m \cdot s)$ by induction hypothesis (such that $v \wr v_m$).

(e) $\Delta, v_{m_2} \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \Rightarrow (\Delta, v_m \cdot s)$ by definition using x.d.

(f) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Rightarrow (\Delta, v_m \cdot s)$ by Lemma A1 on x.c and x.e.

(xi) $d = d_1\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1], \quad \Omega \vdash d_2 \Rightarrow v_2, \quad v_2 \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \Rightarrow v,$
$\Omega \vdash d_1\ d_2 \Rightarrow v, \quad d_1\ d_2 \Downarrow c, \quad \Omega \wr \Delta$. Using the $d_1\ d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad c = c_1 \cdot c_2 \cdot \mathsf{IApp}$.

We are now in position to prove the main goal $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Rightarrow (\Delta, m_v \cdot s)$.

(a) $d \Downarrow c, \quad \Omega_1 \wr \Delta_1, \quad \Delta, s \vdash c_1 \Rightarrow (\Delta, c[\Delta_1]_{rec} \cdot s)$ by induction hypothesis (such that $(\mu.\lambda.d)[\Omega_1] \mapsdown c[\Delta_1]_{rec}$).

(b) $\Delta, c[\Delta_1]_{rec} \cdot s \vdash c_2 \Rightarrow (\Delta, v_{m_2} \cdot c[\Delta_1]_{rec} \cdot s)$ by induction hypothesis (such that $v_2 \mapsdown v_{m_2}$).

(c) $\Delta, s \vdash c_1 \cdot c_2 \Rightarrow (\Delta, v_{m_2} \cdot c[\Delta_1]_{rec} \cdot s)$ by Lemma A1 on xi.a and xi.b.

(d) $v_{m_2} \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \Rightarrow (v_{m_2} \cdot c[\Delta_1]_{rec} \cdot \Delta_1, v_m \cdot s)$ by induction hypothesis (such that $v \mapsdown v_m$).

(e) $\Delta, v_{m_2} \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \Rightarrow (\Delta, v_m \cdot s)$ by definition using xi.d.

(f) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \Rightarrow (\Delta, v_m \cdot s)$ by Lemma A1 on xi.c and xi.e. $\square$

Lemma A2. *Let m be a machine configuration, n any natural number,*

$$m \xrightarrow{\infty} \quad \textit{if and only if} \quad m \xrightarrow[n]{\infty}$$

Proof.

(i) If $m \xrightarrow{\infty}$ then $m \xrightarrow[n]{\infty}$. By coinduction.

Hypothesis $m \xrightarrow{\infty}$. Using the $m \xrightarrow{\infty}$ hypothesis by $\xrightarrow{\infty}$ definition necessarily $m \to m_1$, $m_1 \xrightarrow{\infty}$. We are now in position to prove $m \xrightarrow[n]{\infty}$.

(a) $m_1 \xrightarrow[n]{\infty}$ by coinduction hypothesis on $m_1 \xrightarrow{\infty}$.

(b) $m \xrightarrow[n]{\infty}$ by the $\xrightarrow[n]{\infty}$-perform rule on $m \to m_1$ and $m_1 \xrightarrow[n]{\infty}$.

(ii) If $m \xrightarrow[n]{\infty}$ then $m \xrightarrow{\infty}$. By coinduction.

Hypothesis $m \xrightarrow[n]{\infty}$. We have to prove $m \xrightarrow{\infty}$.

(a) The proof uses the fact that if $m \xrightarrow[n]{\infty}$ then there exists m_1 such that $m \to m_1$ and $m_1 \xrightarrow[n_1]{\infty}$ which is proved by induction on the natural n.

(b) $m \to m_1, \quad m_1 \xrightarrow[n_1]{\infty}$ by ii.a on $m \xrightarrow[n]{\infty}$.

(c) $m_1 \xrightarrow{\infty}$ by coinduction hypothesis on $m_1 \xrightarrow[n_1]{\infty}$.

(d) $m \xrightarrow{\infty}$ by $\xrightarrow{\infty}$ definition on $m \to m_1$ and $m_1 \xrightarrow{\infty}$. $\square$

Lemma A3. *If* $\Omega \vdash d \overset{\infty}{\Rightarrow}$ *, then for all codes* c*, for all stacks* s*,* $([\![d]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow[\|d\|]{\infty}$.

Proof. By coinduction.

(i) $d = d_1 \star d_2$.

Hypothesis $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$. We have to prove $([\![d_1 \star d_2]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow[\|d_1 \star d_2\|]{\infty}$.

(a) for all c_h, s_h, $([\![d_1]\!] \cdot c_h, [\![\Omega]\!], s_h) \xrightarrow[\|d_1\|]{\infty}$ by coinduction hypothesis on $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

(b) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IOp} \cdot c, [\![\Omega]\!], s) \xrightarrow[\|d_1\|]{\infty}$ by i.a with $c_h = [\![d_2]\!] \cdot \mathsf{IOp} \cdot c$ and $s_h = s$.

(c) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IOp} \cdot c, [\![\Omega]\!], s) \xrightarrow[\|d_1\|+1]{\infty}$ by $\xrightarrow[n]{\infty}$-sleep rule on i.b.

(d) $([\![d_1 \star d_2]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow[\|d_1 \star d_2\|]{\infty}$ by definition $[\![d_1 \star d_2]\!] = [\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IOp}$ and $\|d_1 \star d_2\| = \|d_1\| + 1$.

(ii) $d = d_1 \star d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow n_1$, $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$. We must prove $([\![d_1 \star d_2]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow[\|d_1 \star d_2\|]{\infty}$.

(a) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IOp} \cdot c, [\![\Omega]\!], s) \overset{+}{\rightarrow} ([\![d_2]\!] \cdot \mathsf{IOp} \cdot c, [\![\Omega]\!], [\![n_1]\!] \cdot s)$ by Theorem A1 on $\Omega \vdash d_1 \Rightarrow n_1$.

(b) for all c_h, s_h, $([\![d_2]\!] \cdot c_h, [\![\Omega]\!], s_h) \xrightarrow[\|d_2\|]{\infty}$ by induction hypothesis on $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

(c) $([\![d_2]\!] \cdot \mathsf{IOp} \cdot c, [\![\Omega]\!], [\![n_1]\!] \cdot s) \xrightarrow[\|d_2\|]{\infty}$ by ii.b with $c_h = \mathsf{IOp} \cdot c$ and $s_h = [\![n_1]\!] \cdot s$.

(d) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IOp} \cdot c, [\![\Omega]\!], s) \xrightarrow[\|d_1 \star d_2\|]{\infty}$ by $\xrightarrow[n]{\infty}$-perform rule on ii.a and ii.c.

(e) $([\![d_1 \star d_2]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow[\|d_1 \star d_2\|]{\infty}$ by definition $[\![d_1 \star d_2]\!] = [\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IOp}$.

(iii) $d = \mathtt{let}\ d_1\ \mathtt{in}\ d_2$.

Hypothesis $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$. We have to prove $([\![\mathtt{let}\ d_1\ \mathtt{in}\ d_2]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow[\|\mathtt{let}\ d_1\ \mathtt{in}\ d_2\|]{\infty}$.

(a) for all c_h, s_h, $([\![d_1]\!] \cdot c_h, [\![\Omega]\!], s_h) \xrightarrow[\|d_1\|]{\infty}$ by coinduction hypothesis on $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

(b) $([\![d_1]\!] \cdot \mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c, [\![\Omega]\!], s) \xrightarrow[\|d_1\|]{\infty}$ by iii.a with $c_h = \mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c$ and $s_h = s$.

(c) $([\![d_1]\!] \cdot \mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c, [\![\Omega]\!], s) \xrightarrow[\|d_1\|+1]{\infty}$ by $\xrightarrow[n]{\infty}$-sleep rule on iii.b.

(d) $([\![\mathtt{let}\ d_1\ \mathtt{in}\ d_2]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow[\|\mathtt{let}\ d_1\ \mathtt{in}\ d_2\|]{\infty}$
by definition $[\![\mathtt{let}\ d_1\ \mathtt{in}\ d_2]\!] = [\![d_1]\!] \cdot \mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet}$ and $\|\mathtt{let}\ d_1\ \mathtt{in}\ d_2\| = \|d_1\| + 1$.

(iv) $d = \mathtt{let}\ d_1\ \mathtt{in}\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow v_1$, $v_1 \cdot \Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

We must prove $([\![\mathtt{let}\ d_1\ \mathtt{in}\ d_2]\!] \cdot c, [\![\Omega]\!], s) \xrightarrow[\|\mathtt{let}\ d_1\ \mathtt{in}\ d_2\|]{\infty}$.

(a) $([\![d_1]\!] \cdot \mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c, [\![\Omega]\!], s) \overset{+}{\rightarrow} (\mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c, [\![\Omega]\!], [\![v_1]\!] \cdot s)$ by Theorem A1 on $\Omega \vdash d_1 \Rightarrow v_1$.

(b) $(\mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c, [\![\Omega]\!], [\![v_1]\!] \cdot s) \rightarrow ([\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c, [\![v_1]\!] \cdot [\![\Omega]\!], s)$ by $\rightarrow$ definition of ILet.

(c) for all c_h, s_h, $([\![d_2]\!] \cdot c_h, [\![v_1]\!] \cdot [\![\Omega]\!], s_h) \overset{\infty}{\underset{\|d_2\|}{\rightarrow}}$ by coinduction hypothesis on $v_1 \cdot \Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

(d) $([\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c, [\![v_1]\!] \cdot [\![\Omega]\!], s) \overset{\infty}{\underset{\|d_2\|}{\rightarrow}}$ by iv.c with $c_h = \mathsf{IEndLet} \cdot c$ and $s_h = s$.

(e) $(\mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c, [\![\Omega]\!], [\![v_1]\!] \cdot s) \overset{\infty}{\underset{\|d_1\|}{\rightarrow}}$ by $\overset{\infty}{\underset{n}{\rightarrow}}$-perform rule on iv.b and iv.d.

(f) $([\![d_1]\!] \cdot \mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet} \cdot c, [\![\Omega]\!], s) \overset{\infty}{\underset{\|d_1\|+1}{\rightarrow}}$ by $\overset{\infty}{\underset{n}{\rightarrow}}$-perform rule on iv.a and iv.e.

(g) $([\![\mathtt{let}\ d_1\ \mathtt{in}\ d_2]\!] \cdot c, [\![\Omega]\!], s) \overset{\infty}{\underset{\|\mathtt{let}\ d_1\ \mathtt{in}\ d_2\|}{\rightarrow}}$
by definition $[\![\mathtt{let}\ d_1\ \mathtt{in}\ d_2]\!] = [\![d_1]\!] \cdot \mathsf{ILet} \cdot [\![d_2]\!] \cdot \mathsf{IEndLet}$ and $\|\mathtt{let}\ d_1\ \mathtt{in}\ d_2\| = \|d_1\| + 1$.

(v) $d = \mathtt{if}\ d_1\ \mathtt{then}\ d_2\ \mathtt{else}\ d_3$.

Hypothesis $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$. We must prove $([\![\mathtt{if}\ d_1\ \mathtt{then}\ d_2\ \mathtt{else}\ d_3]\!] \cdot c, [\![\Omega]\!], s) \overset{\infty}{\underset{\|\mathtt{if}\ d_1\ \mathtt{then}\ d_2\ \mathtt{else}\ d_3\|}{\rightarrow}}$.

(a) for all c_h, s_h, $([\![d_1]\!] \cdot c_h, [\![\Omega]\!], s_h) \overset{\infty}{\underset{\|d_1\|}{\rightarrow}}$ by coinduction hypothesis on $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

(b) $([\![d_1]\!] \cdot \mathsf{ISel}\ ([\![d_2]\!] \cdot \mathsf{IJoin})\ ([\![d_3]\!] \cdot \mathsf{IJoin}) \cdot c, [\![\Omega]\!], s) \overset{\infty}{\underset{\|d_1\|}{\rightarrow}}$ by v.a with
$c_h = \mathsf{ISel}\ ([\![d_2]\!] \cdot \mathsf{IJoin})\ ([\![d_3]\!] \cdot \mathsf{IJoin}) \cdot c$ and $s_h = s$.

(c) $([\![d_1]\!] \cdot \mathsf{ISel}\ ([\![d_2]\!] \cdot \mathsf{IJoin})\ ([\![d_3]\!] \cdot \mathsf{IJoin}) \cdot c, [\![\Omega]\!], s) \overset{\infty}{\underset{\|d_1\|+1}{\rightarrow}}$ by $\overset{\infty}{\underset{n}{\rightarrow}}$-sleep rule on v.b.

(d) $([\![\mathtt{if}\ d_1\ \mathtt{then}\ d_2\ \mathtt{else}\ d_3]\!] \cdot c, [\![\Omega]\!], s) \overset{\infty}{\underset{\|\mathtt{if}\ d_1\ \mathtt{then}\ d_2\ \mathtt{else}\ d_3\|}{\rightarrow}}$ by definition
$[\![\mathtt{if}\ d_1\ \mathtt{then}\ d_2\ \mathtt{else}\ d_3]\!] = [\![d_1]\!] \cdot \mathsf{ISel}\ ([\![d_2]\!] \cdot \mathsf{IJoin})\ ([\![d_3]\!] \cdot \mathsf{IJoin})$ and
$\|\mathtt{if}\ d_1\ \mathtt{then}\ d_2\ \mathtt{else}\ d_3\| = \|d_1\| + 1$.

(vi) $d = \mathtt{if}\ d_1\ \mathtt{then}\ d_2\ \mathtt{else}\ d_3$.

Hypothesis $\Omega \vdash d_1 \Rightarrow \mathit{true}, \quad \Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

We must prove $([\![\mathtt{if}\ d_1\ \mathtt{then}\ d_2\ \mathtt{else}\ d_3]\!] \cdot c, [\![\Omega]\!], s) \overset{\infty}{\underset{\|\mathtt{if}\ d_1\ \mathtt{then}\ d_2\ \mathtt{else}\ d_3\|}{\rightarrow}}$.

(a) $([\![d_1]\!] \cdot \mathsf{ISel}\ ([\![d_2]\!] \cdot \mathsf{IJoin})\ ([\![d_3]\!] \cdot \mathsf{IJoin}) \cdot c, [\![\Omega]\!], s) \overset{+}{\rightarrow}$
$(\mathsf{ISel}\ ([\![d_2]\!] \cdot \mathsf{IJoin})\ ([\![d_3]\!] \cdot \mathsf{IJoin}) \cdot c, [\![\Omega]\!], \mathit{true} \cdot s)$ by Theorem A1 on $\Omega \vdash d_1 \Rightarrow \mathit{true}$.

(b) $(\mathsf{ISel}\ ([\![d_2]\!] \cdot \mathsf{IJoin})\ ([\![d_3]\!] \cdot \mathsf{IJoin}) \cdot c, [\![\Omega]\!], \mathit{true} \cdot s) \rightarrow ([\![d_2]\!] \cdot \mathsf{IJoin}, [\![\Omega]\!], (c, []) \cdot s)$ by $\rightarrow$ definition of ISel.

(c) for all c_h, s_h, $([\![d_2]\!] \cdot c_h, [\![\Omega]\!], s_h) \overset{\infty}{\underset{\|d_2\|}{\rightarrow}}$ by coinduction hypothesis on $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

(d) $([\![d_2]\!] \cdot \mathsf{IJoin}, [\![\Omega]\!], (c, []) \cdot s) \overset{\infty}{\underset{\|d_2\|}{\rightarrow}}$ by vi.c with $c_h = \mathsf{IJoin}$ and $s_h = (c, []) \cdot s$.

(e) $(\mathsf{ISel}\ ([\![d_2]\!] \cdot \mathsf{IJoin})\ ([\![d_3]\!] \cdot \mathsf{IJoin}) \cdot c, [\![\Omega]\!], \mathit{true} \cdot s) \overset{\infty}{\underset{\|d_1\|}{\rightarrow}}$ by $\overset{\infty}{\underset{n}{\rightarrow}}$-perform rule on vi.b and vi.d.

(f) $([\![d_1]\!] \cdot \mathsf{ISel}\ ([\![d_2]\!] \cdot \mathsf{IJoin})\ ([\![d_3]\!] \cdot \mathsf{IJoin}) \cdot c, [\![\Omega]\!], s) \overset{\infty}{\underset{\|d_1\|+1}{\rightarrow}}$ by $\overset{\infty}{\underset{n}{\rightarrow}}$-perform rule on vi.a and vi.e.

(g) $(\llbracket \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\|}{\Rightarrow}}$ by definition $\llbracket \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin})$ and $\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\| = \|d_1\| + 1$.

(vii) $d = \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3$.

Hypothesis $\Omega \vdash d_1 \Rightarrow \mathit{false}, \quad \Omega \vdash d_3 \overset{\infty}{\Rightarrow}$.

We must prove $(\llbracket \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\|}{\Rightarrow}}$.

(a) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\to}$ $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, \mathit{false} \cdot s)$ by Theorem A1 on $\Omega \vdash d_1 \Rightarrow \mathit{false}$.

(b) $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, \mathit{false} \cdot s) \to (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, []) \cdot s)$ by $\to$ definition of ISel.

(c) for all c_h, s_h, $(\llbracket d_3 \rrbracket \cdot c_h, \llbracket \Omega \rrbracket, s_h) \overset{\infty}{\underset{\|d_3\|}{\Rightarrow}}$ by coinduction hypothesis on $\Omega \vdash d_3 \overset{\infty}{\Rightarrow}$.

(d) $(\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}, \llbracket \Omega \rrbracket, (c, []) \cdot s) \overset{\infty}{\underset{\|d_3\|}{\Rightarrow}}$ by vii.c with $c_h = \mathsf{IJoin}$ and $s_h = (c, []) \cdot s$.

(e) $(\mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, \mathit{false} \cdot s) \overset{\infty}{\underset{\|d_1\|}{\Rightarrow}}$ by $\overset{\infty}{\underset{n}{\Rightarrow}}$-perform rule on vii.b and vii.d.

(f) $(\llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin}) \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\|+1}{\Rightarrow}}$ by $\overset{\infty}{\underset{n}{\Rightarrow}}$-perform rule on vii.a and vii.e.

(g) $(\llbracket \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\|}{\Rightarrow}}$ by definition $\llbracket \texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3 \rrbracket = \llbracket d_1 \rrbracket \cdot \mathsf{ISel}\ (\llbracket d_2 \rrbracket \cdot \mathsf{IJoin})\ (\llbracket d_3 \rrbracket \cdot \mathsf{IJoin})$ and $\|\texttt{if } d_1 \texttt{ then } d_2 \texttt{ else } d_3\| = \|d_1\| + 1$.

(viii) $d = d_1\ d_2$.

Hypothesis $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$. We must prove $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\ d_2\|}{\Rightarrow}}$.

(a) for all c_h, s_h, $(\llbracket d_1 \rrbracket \cdot c_h, \llbracket \Omega \rrbracket, s_h) \overset{\infty}{\underset{\|d_1\|}{\Rightarrow}}$ by coinduction hypothesis on $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

(b) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\|}{\Rightarrow}}$ by viii.a with $c_h = \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c$ and $s_h = s$.

(c) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\|+1}{\Rightarrow}}$ by $\overset{\infty}{\underset{n}{\Rightarrow}}$-sleep rule on viii.b.

(d) $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\ d_2\|}{\Rightarrow}}$ by definition $\llbracket d_1\ d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp}$ and $\|d_1\ d_2\| = \|d_1\| + 1$.

(ix) $d = d_1\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1], \quad \Omega \vdash d_2 \overset{\infty}{\Rightarrow}$. We must prove $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\underset{\|d_1\ d_2\|}{\Rightarrow}}$.

(a) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\to} (\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s)$ by Theorem A1 on $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$.

(b) for all c_h, s_h, $(\llbracket d_2 \rrbracket \cdot c_h, \llbracket \Omega \rrbracket, s_h) \overset{\infty}{\underset{\|d_2\|}{\Rightarrow}}$ by coinduction hypothesis on $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

(c) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s) \overset{\infty}{\underset{\|d_2\|}{\Rightarrow}}$ by ix.b with $c_h = \mathsf{IApp} \cdot c$ and $s_h = \llbracket (\lambda.d)[\Omega_1] \rrbracket \cdot s$.

(d) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], s) \; \underset{\|d_1\|+1}{\overset{\infty}{\Rightarrow}}$ by $\underset{n}{\overset{\infty}{\Rightarrow}}$-perform rule on ix.a and ix.c.

(e) $([\![d_1\ d_2]\!] \cdot c, [\![\Omega]\!], s) \; \underset{\|d_1\ d_2\|}{\overset{\infty}{\Rightarrow}}$ by definition $[\![d_1\ d_2]\!] = [\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp}$ and $\|d_1\ d_2\| = \|d_1\| + 1$.

(x) $d = d_1\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$, $\quad \Omega \vdash d_2 \overset{\infty}{\Rightarrow}$. We must prove $([\![d_1\ d_2]\!] \cdot c, [\![\Omega]\!], s) \; \underset{\|d_1\ d_2\|}{\overset{\infty}{\Rightarrow}}$.

(a) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], s) \overset{+}{\rightarrow} ([\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], [\![(\mu.\lambda.d)[\Omega_1]]\!] \cdot s)$ by Theorem A1 on $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$.

(b) for all c_h, s_h, $([\![d_2]\!] \cdot c_h, [\![\Omega]\!], s_h) \; \underset{\|d_2\|}{\overset{\infty}{\Rightarrow}}$ by coinduction hypothesis on $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

(c) $([\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], [\![(\mu.\lambda.d)[\Omega_1]]\!] \cdot s) \; \underset{\|d_2\|}{\overset{\infty}{\Rightarrow}}$ by x.b with $c_h = \mathsf{IApp} \cdot c$ and $s_h = [\![(\mu.\lambda.d)[\Omega_1]]\!] \cdot s$.

(d) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], s) \; \underset{\|d_1\|+1}{\overset{\infty}{\Rightarrow}}$ by $\underset{n}{\overset{\infty}{\Rightarrow}}$-perform rule on x.a and x.c.

(e) $([\![d_1\ d_2]\!] \cdot c, [\![\Omega]\!], s) \; \underset{\|d_1\ d_2\|}{\overset{\infty}{\Rightarrow}}$ by definition $[\![d_1\ d_2]\!] = [\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp}$ and $\|d_1\ d_2\| = \|d_1\| + 1$.

(xi) $d = d_1\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$, $\quad \Omega \vdash d_2 \Rightarrow v_2$, $\quad v_2 \cdot \Omega_1 \vdash d \overset{\infty}{\Rightarrow}$.
We must prove $([\![d_1\ d_2]\!] \cdot c, [\![\Omega]\!], s) \; \underset{\|d_1\ d_2\|}{\overset{\infty}{\Rightarrow}}$.

(a) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], s) \overset{+}{\rightarrow} ([\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], [\![(\lambda.d)[\Omega_1]]\!] \cdot s)$ by Theorem A1 on $\Omega \vdash d_1 \Rightarrow (\lambda.d)[\Omega_1]$.

(b) $([\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], [\![(\lambda.d)[\Omega_1]]\!] \cdot s) \overset{+}{\rightarrow} (\mathsf{IApp} \cdot c, [\![\Omega]\!], [\![v_2]\!] \cdot [\![(\lambda.d)[\Omega_1]]\!] \cdot s)$ by Theorem A1 on $\Omega \vdash d_2 \Rightarrow v_2$.

(c) $(\mathsf{IApp} \cdot c, [\![\Omega]\!], [\![v_2]\!] \cdot ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]] \cdot s) \rightarrow ([\![d]\!] \cdot \mathsf{IRet}, [\![v_2]\!] \cdot [\![\Omega_1]\!], (c, [\![\Omega]\!]) \cdot s)$ by $\rightarrow$ of IApp.

(d) for all c_h, s_h, $([\![d]\!] \cdot c_h, [\![v_2]\!] \cdot [\![\Omega_1]\!], s_h) \; \underset{\|d\|}{\overset{\infty}{\Rightarrow}}$ by coinduction hypothesis on $v_2 \cdot \Omega_1 \vdash d \overset{\infty}{\Rightarrow}$.

(e) $([\![d]\!] \cdot \mathsf{IRet}, [\![v_2]\!] \cdot [\![\Omega_1]\!], (c, [\![\Omega]\!]) \cdot s) \; \underset{\|d\|}{\overset{\infty}{\Rightarrow}}$ by xi.d with $c_h = \mathsf{IRet}$ and $s_h = (c, [\![\Omega]\!]) \cdot s$.

(f) $(\mathsf{IApp} \cdot c, [\![\Omega]\!], [\![v_2]\!] \cdot ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]] \cdot s) \; \underset{\|d_2\|}{\overset{\infty}{\Rightarrow}}$ by $\underset{n}{\overset{\infty}{\Rightarrow}}$-perform rule on xi.c and xi.e.

(g) $([\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], [\![(\lambda.d)[\Omega_1]]\!] \cdot s) \; \underset{\|d_1\|}{\overset{\infty}{\Rightarrow}}$ by $\underset{n}{\overset{\infty}{\Rightarrow}}$-perform rule on xi.b and xi.f, and by definition $[\![(\lambda.d)[\Omega_1]]\!] = ([\![d]\!] \cdot \mathsf{IRet})[[\![\Omega_1]\!]]$.

(h) $([\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp} \cdot c, [\![\Omega]\!], s) \; \underset{\|d_1\|+1}{\overset{\infty}{\Rightarrow}}$ by $\underset{n}{\overset{\infty}{\Rightarrow}}$-perform rule on xi.a and xi.g.

(i) $([\![d_1\ d_2]\!] \cdot c, [\![\Omega]\!], s) \; \underset{\|d_1\ d_2\|}{\overset{\infty}{\Rightarrow}}$ by definition $[\![d_1\ d_2]\!] = [\![d_1]\!] \cdot [\![d_2]\!] \cdot \mathsf{IApp}$ and $\|d_1\ d_2\| = \|d_1\| + 1$.

(xii) $d = d_1\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$, $\quad \Omega \vdash d_2 \Rightarrow v_2$, $\quad v_2 \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \overset{\infty}{\Rightarrow}$.
We must prove $([\![d_1\ d_2]\!] \cdot c, [\![\Omega]\!], s) \; \underset{\|d_1\ d_2\|}{\overset{\infty}{\Rightarrow}}$.

(a) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \overset{+}{\rightarrow} (\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s)$ by Theorem A1 on $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d)[\Omega_1]$.

(b) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s) \overset{+}{\rightarrow} (\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s)$ by Theorem A1 on $\Omega \vdash d_2 \Rightarrow v_2$.

(c) $(\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot s) \rightarrow$
$(\llbracket d \rrbracket \cdot \mathsf{IRet}, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot \llbracket \Omega_1 \rrbracket, (c, \llbracket \Omega \rrbracket) \cdot s)$ by $\rightarrow$ of IApp.

(d) for all c_h, s_h, $(\llbracket d \rrbracket \cdot c_h, \llbracket v_2 \rrbracket \cdot \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot \llbracket \Omega_1 \rrbracket, s_h) \underset{\|d\|}{\overset{\infty}{\rightarrow}}$ by coinduction hypothesis on $v_2 \cdot (\mu.\lambda.d)[\Omega_1] \cdot \Omega_1 \vdash d \overset{\infty}{\Rrightarrow}$.

(e) $(\llbracket d \rrbracket \cdot \mathsf{IRet}, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot \llbracket \Omega_1 \rrbracket, (c, \llbracket \Omega \rrbracket) \cdot s) \underset{\|d\|}{\overset{\infty}{\rightarrow}}$ by xii.d with $c_h = \mathsf{IRet}$ and $s_h = (c, \llbracket \Omega \rrbracket) \cdot s$, and by definition $\llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket = (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec}$.

(f) $(\mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket v_2 \rrbracket \cdot (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec} \cdot s) \underset{\|d_2\|}{\overset{\infty}{\rightarrow}}$ by $\underset{n}{\overset{\infty}{\rightarrow}}$-perform rule on xii.c and xii.e.

(g) $(\llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, \llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket \cdot s) \underset{\|d_1\|}{\overset{\infty}{\rightarrow}}$ by $\underset{n}{\overset{\infty}{\rightarrow}}$-perform rule on xii.b and xii.f, and by definition $\llbracket (\mu.\lambda.d)[\Omega_1] \rrbracket = (\llbracket d \rrbracket \cdot \mathsf{IRet})[\llbracket \Omega_1 \rrbracket]_{rec}$.

(h) $(\llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp} \cdot c, \llbracket \Omega \rrbracket, s) \underset{\|d_1\|+1}{\overset{\infty}{\rightarrow}}$ by $\underset{n}{\overset{\infty}{\rightarrow}}$-perform rule on xii.a and xii.g.

(i) $(\llbracket d_1\ d_2 \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\|d_1\ d_2\|}{\overset{\infty}{\rightarrow}}$ by definition $\llbracket d_1\ d_2 \rrbracket = \llbracket d_1 \rrbracket \cdot \llbracket d_2 \rrbracket \cdot \mathsf{IApp}$ and $\|d_1\ d_2\| = \|d_1\| + 1$. $\square$

Theorem A3. *If $\Omega \vdash d \overset{\infty}{\Rrightarrow}$, then $(\llbracket d \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\rightarrow}$ for all codes c and stacks s.*

Proof. Hypothesis $\Omega \vdash d \overset{\infty}{\Rrightarrow}$. We must prove $(\llbracket d \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\rightarrow}$.

(i) $(\llbracket d \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \underset{\|d\|}{\overset{\infty}{\rightarrow}}$ by Lemma A3 on $\Omega \vdash d \overset{\infty}{\Rrightarrow}$.

(ii) $(\llbracket d \rrbracket \cdot c, \llbracket \Omega \rrbracket, s) \overset{\infty}{\rightarrow}$ by Lemma A2 on i. $\square$

Lemma A4. *Let Δ be a machine environment, s a stack, i a machine instruction,*

$$\Delta, s \vdash i \overset{\infty}{\Rrightarrow} \qquad \textit{if and only if} \qquad \Delta, s \vdash i \overset{\infty}{\Mapsto}$$

and, let c be a machine code, n any natural,

$$\Delta, s \vdash c \overset{\infty}{\Rrightarrow} \qquad \textit{if and only if} \qquad \Delta, s \vdash c \underset{n}{\overset{\infty}{\Mapsto}}$$

Proof.

1. If $\Delta, s \vdash i \overset{\infty}{\Rrightarrow}$ then $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ and if $\Delta, s \vdash c \overset{\infty}{\Rrightarrow}$ then $\Delta, s \vdash c \underset{n}{\overset{\infty}{\Mapsto}}$.

   (a) If $\Delta, s \vdash i \overset{\infty}{\Rrightarrow}$ then $\Delta, s \vdash i \overset{\infty}{\Mapsto}$. Assuming 1b, the proof is by case analysis.

      i. $i = \mathsf{ISel}\ c_1\ c_2$.

      Hypothesis $\Delta, s \vdash c_1 \overset{\infty}{\Rrightarrow}$, $\quad \Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Rrightarrow}$.
      We must prove $\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Mapsto}$.

A. $\Delta, s \vdash c_1 \ \underset{n}{\overset{\infty}{\Mapsto}}$ by 1b on $\Delta, s \vdash c_1 \overset{\infty}{\Rightarrow}$.

B. $\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Mapsto}$ by definition using 1(a)iA.

ii. $i = \mathsf{ISel}\ c_1\ c_2$.

Hypothesis $\Delta, s \vdash c_1 \overset{\infty}{\Rightarrow}$, $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Rightarrow}$.
We must prove $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Mapsto}$.

A. $\Delta, s \vdash c_1 \ \underset{n}{\overset{\infty}{\Mapsto}}$ by 1b on $\Delta, s \vdash c_1 \overset{\infty}{\Rightarrow}$.

B. $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Mapsto}$ by definition using 1(a)iiA.

iii. $i = \mathsf{IApp}$.

Hypothesis $v \cdot \Delta_1, s \vdash c \overset{\infty}{\Rightarrow}$, $\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Rightarrow}$.
We must prove $\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Mapsto}$.

A. $v \cdot \Delta_1, s \vdash c \ \underset{n}{\overset{\infty}{\Mapsto}}$ by 1b on $v \cdot \Delta_1, s \vdash c \overset{\infty}{\Rightarrow}$.

B. $\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Mapsto}$ by definition using 1(a)iiiA.

iv. $i = \mathsf{IApp}$.

Hypothesis $v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \overset{\infty}{\Rightarrow}$, $\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Rightarrow}$.
We must prove $\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Mapsto}$.

A. $v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \ \underset{n}{\overset{\infty}{\Mapsto}}$ by 1b on $v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \overset{\infty}{\Rightarrow}$.

B. $\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Mapsto}$ by definition using 1(a)ivA.

(b) If $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$ then $\Delta, s \vdash c \ \underset{n}{\overset{\infty}{\Mapsto}}$. By coinduction.

i. $c = i \cdot c'$.

Hypothesis $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$, $\Delta, s \vdash i \cdot c' \overset{\infty}{\Rightarrow}$. We have to prove $\Delta, s \vdash i \cdot c' \ \underset{n}{\overset{\infty}{\Mapsto}}$.

A. $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ by 1a on $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$.

B. $\Delta, s \vdash i \cdot c' \ \underset{n}{\overset{\infty}{\Mapsto}}$ by definition using 1(b)iA.

ii. $c = i \cdot c'$.

Hypothesis $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$, $\Delta_1, s_1 \vdash c' \overset{\infty}{\Rightarrow}$, $\Delta, s \vdash i \cdot c' \overset{\infty}{\Rightarrow}$.
We have to prove $\Delta, s \vdash i \cdot c' \ \underset{n}{\overset{\infty}{\Mapsto}}$.

A. $\Delta_1, s_1 \vdash c' \ \underset{n}{\overset{\infty}{\Mapsto}}$ by coinduction hypothesis on $\Delta_1, s_1 \vdash c' \overset{\infty}{\Rightarrow}$.

B. $\Delta, s \vdash i \cdot c' \ \underset{n}{\overset{\infty}{\Mapsto}}$ by $\underset{n}{\overset{\infty}{\Mapsto}}$-perform rule on $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ and 1(b)iiA.

2. If $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ then $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$ and if $\Delta, s \vdash c \ \underset{n}{\overset{\infty}{\Mapsto}}$ then $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$.

(a) If $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ then $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$. Assuming 2b, the proof is by case analysis.

i. $i = \mathsf{ISel}\ c_1\ c_2$.

Hypothesis $\Delta, s \vdash c_1\ \underset{n}{\overset{\infty}{\Mapsto}}\ ,\quad \Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Mapsto}$.
We must prove $\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Rightarrow}$.

A. $\Delta, s \vdash c_1 \overset{\infty}{\Rightarrow}$ by 2b on $\Delta, s \vdash c_1\ \underset{n}{\overset{\infty}{\Mapsto}}$.

B. $\Delta, true \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Rightarrow}$ by definition using 2(a)iA.

ii. $i = \mathsf{ISel}\ c_1\ c_2$.

Hypothesis $\Delta, s \vdash c_1\ \underset{n}{\overset{\infty}{\Mapsto}}\ ,\quad \Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Mapsto}$.
We must prove $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Rightarrow}$.

A. $\Delta, s \vdash c_1 \overset{\infty}{\Rightarrow}$ by 2b on $\Delta, s \vdash c_1\ \underset{n}{\overset{\infty}{\Mapsto}}$.

B. $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_1\ c_2 \overset{\infty}{\Rightarrow}$ by definition using 2(a)iiA.

iii. $i = \mathsf{IApp}$.

Hypothesis $v \cdot \Delta_1, s \vdash c\ \underset{n}{\overset{\infty}{\Mapsto}}\ ,\quad \Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Mapsto}$.
We must prove $\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Rightarrow}$.

A. $v \cdot \Delta_1, s \vdash c \overset{\infty}{\Rightarrow}$ by 2b on $v \cdot \Delta_1, s \vdash c\ \underset{n}{\overset{\infty}{\Mapsto}}$.

B. $\Delta, v \cdot c[\Delta_1] \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Rightarrow}$ by definition using 2(a)iiiA.

iv. $i = \mathsf{IApp}$.

Hypothesis $v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c\ \underset{n}{\overset{\infty}{\Mapsto}}\ ,\quad \Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Mapsto}$.
We must prove $\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Rightarrow}$.

A. $v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c \overset{\infty}{\Rightarrow}$ by 2b on $v \cdot c[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c\ \underset{n}{\overset{\infty}{\Mapsto}}$.

B. $\Delta, v \cdot c[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \overset{\infty}{\Rightarrow}$ by definition using 2(a)ivA.

(b) If $\Delta, s \vdash c\ \underset{n}{\overset{\infty}{\Mapsto}}$ then $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$. By coinduction.

i. $c = i \cdot c$.

Hypothesis $\Delta, s \vdash i \overset{\infty}{\Mapsto},\quad \Delta, s \vdash i \cdot c\ \underset{n}{\overset{\infty}{\Mapsto}}$. We must prove $\Delta, s \vdash i \cdot c \overset{\infty}{\Rightarrow}$.

A. $\Delta, s \vdash i \overset{\infty}{\Rightarrow}$ by 2a on $\Delta, s \vdash i \overset{\infty}{\Mapsto}$.

B. $\Delta, s \vdash i \cdot c \overset{\infty}{\Rightarrow}$ by definition using 2(b)iA.

ii. $c = c_1 \cdot c_2$.

Hypothesis $\Delta, s \vdash c_1\ \underset{n}{\overset{\infty}{\Mapsto}}\ ,\quad \Delta, s \vdash c_1 \cdot c_2 \underset{n+1}{\overset{\infty}{\Mapsto}}$. We must prove $\Delta, s \vdash c_1 \cdot c_2 \overset{\infty}{\Rightarrow}$.

A. The proof uses the fact that if $\Delta, s \vdash i \cdot c\ \underset{n}{\overset{\infty}{\Mapsto}}$ then $\Delta, s \vdash i \overset{\infty}{\Mapsto}$ or there exists n_1, Δ_1, s_1, such that $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ and $\Delta_1, s_1 \vdash c\ \underset{n_1}{\overset{\infty}{\Mapsto}}$, which is proved by induction on the natural n.

B. $c_1 \neq []$ that is $c_1 = i \cdot c_1'$ by $\underset{n}{\overset{\infty}{\Mapsto}}$ definition and $\Delta, s \vdash c_1 \underset{n}{\overset{\infty}{\Mapsto}}$.

C. $\Delta, s \vdash i \overset{\infty}{\Mapsto}$, or $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ and $\Delta_1, s_1 \vdash c_1' \cdot c_2 \underset{n_1}{\overset{\infty}{\Mapsto}}$ by 2(b)iiA on $\Delta, s \vdash i \cdot c_1' \cdot c_2 \underset{n+1}{\overset{\infty}{\Mapsto}}$.

- $\Delta, s \vdash i \overset{\infty}{\Mapsto}$.
  - $\Delta, s \vdash i \overset{\infty}{\Rrightarrow}$ by 2a on $\Delta, s \vdash i \overset{\infty}{\Mapsto}$.
  - $\Delta, s \vdash i \cdot c_1' \cdot c_2 \overset{\infty}{\Rrightarrow}$ by definition using $\Delta, s \vdash i \overset{\infty}{\Rrightarrow}$.
- $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ and $\Delta_1, s_1 \vdash c_1' \cdot c_2 \underset{n_1}{\overset{\infty}{\Mapsto}}$.
  - $\Delta_1, s_1 \vdash c_1' \cdot c_2 \overset{\infty}{\Rrightarrow}$ by coinduction hypothesis on $\Delta_1, s_1 \vdash c_1' \cdot c_2 \underset{n_1}{\overset{\infty}{\Mapsto}}$.
  - $\Delta, s \vdash i \cdot c_1' \cdot c_2 \overset{\infty}{\Rrightarrow}$ by definition using $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ and $\Delta_1, s_1 \vdash c_1' \cdot c_2 \overset{\infty}{\Rrightarrow}$.

iii. $c = c_1 \cdot c_2$.

Hypothesis $c_1 \neq []$, $\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1)$, $\Delta_1, s_1 \vdash c_2 \underset{n'}{\overset{\infty}{\Mapsto}}$, $\Delta, s \vdash c_1 \cdot c_2 \underset{n}{\overset{\infty}{\Mapsto}}$.
We must prove $\Delta, s \vdash c_1 \cdot c_2 \overset{\infty}{\Rrightarrow}$.

A. $c_1 \neq []$ that is $c_1 = i \cdot c_1'$ by hypothesis.

B. $\Delta, s \vdash i \Rightarrow (\Delta_t, s_t)$ and $\Delta_t, s_t \vdash c_1' \Rightarrow (\Delta_1, s_1)$ by $\Delta, s \vdash c_1 \Rightarrow (\Delta_1, s_1)$ hypothesis.

- $c_1' = []$.
  - $\Delta_t = \Delta_1$ and $s_t = s_1$ by $\Delta_t, s_t \vdash c_1' \Rightarrow (\Delta_1, s_1)$ and $c_1' = []$.
  - $\Delta_1, s_1 \vdash c_2 \overset{\infty}{\Rrightarrow}$ by coinduction hypothesis on $\Delta_1, s_1 \vdash c_2 \underset{n'}{\overset{\infty}{\Mapsto}}$.
  - $\Delta, s \vdash i \cdot c_2 \overset{\infty}{\Rrightarrow}$ by definition using $\Delta, s \vdash i \Rightarrow (\Delta_1, s_1)$ and $\Delta_1, s_1 \vdash c_2 \overset{\infty}{\Rrightarrow}$.
  - $\Delta, s \vdash i \cdot c_1' \cdot c_2 \overset{\infty}{\Rrightarrow}$ by $c_1' = []$.
- $c_1' \neq []$
  - $\Delta_t, s_t \vdash c_1' \cdot c_2 \underset{n'}{\overset{\infty}{\Mapsto}}$ by $\underset{n}{\overset{\infty}{\Mapsto}}$-perform rule on $\Delta_t, s_t \vdash c_1' \Rightarrow (\Delta_1, s_1)$ and $\Delta_1, s_1 \vdash c_2 \underset{n'}{\overset{\infty}{\Mapsto}}$.
  - $\Delta_t, s_t \vdash c_1' \cdot c_2 \overset{\infty}{\Rrightarrow}$ by coinduction hypothesis on $\Delta_t, s_t \vdash c_1' \cdot c_2 \underset{n'}{\overset{\infty}{\Mapsto}}$.
  - $\Delta, s \vdash i \cdot c_1' \cdot c_2 \overset{\infty}{\Rrightarrow}$ by definition using $\Delta, s \vdash i \Rightarrow (\Delta_t, s_t)$ and $\Delta_t, s_t \vdash c_1' \cdot c_2 \overset{\infty}{\Rrightarrow}$. □

Lemma A5 (Correctness for non-termination (auxiliary))**.** *Let Ω be a nameless environment, Δ a machine environment, d a nameless expression, c a machine code. If*

$$\Omega \vdash d \overset{\infty}{\Rrightarrow}, \quad d \Downarrow c, \quad \Omega \wr \Delta$$

then, for all stack s,

$$\Delta, s \vdash c \overset{\infty}{\underset{\|d\|}{\Mapsto}}$$

Proof. By coinduction.

(i) $d = d_1 \star d_2$.

Hypothesis $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}, \quad \Omega \vdash d_1 \star d_2 \overset{\infty}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \wr \Delta$. Using the $d_1 \star d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad c = c_1 \cdot c_2 \cdot \mathsf{IOp}$.
We must prove $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IOp} \overset{\infty}{\underset{\|d_1 \star d_2\|}{\Mapsto}}$.

(a) $\Delta, s \vdash c_1 \overset{\infty}{\underset{\|d_1\|}{\Mapsto}}$ by coinduction hypothesis on $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

(b) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \overset{\infty}{\underset{\|d_1\|+1}{\Mapsto}}$ by $\overset{\infty}{\underset{n}{\Mapsto}}$-sleep rule on i.a.

(c) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \overset{\infty}{\underset{\|d_1 \star d_2\|}{\Mapsto}}$ by definition $\|d_1 \star d_2\| = \|d_1\| + 1$.

(ii) $d = d_1 \star d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow n_1, \quad \Omega \vdash d_2 \overset{\infty}{\Rightarrow}, \quad \Omega \vdash d_1 \star d_2 \overset{\infty}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \wr \Delta$. Using the $d_1 \star d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad c = c_1 \cdot c_2 \cdot \mathsf{IOp}$. We must prove $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IOp} \overset{\infty}{\underset{\|d_1 \star d_2\|}{\Mapsto}}$.

(a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, n_1 \cdot s)$ by Theorem A2 on $\Omega \vdash d_1 \Rightarrow n_1$.

(b) $\Delta, n_1 \cdot s \vdash c_2 \overset{\infty}{\underset{\|d_2\|}{\Mapsto}}$ by coinduction hypothesis on $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

(c) $\Delta, n_1 \cdot s \vdash c_2 \cdot \mathsf{IOp} \overset{\infty}{\underset{\|d_2\|+1}{\Mapsto}}$ by $\overset{\infty}{\underset{n}{\Mapsto}}$-sleep rule on ii.b.

(d) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IOp} \overset{\infty}{\underset{\|d_1\|+1}{\Mapsto}}$ by $\overset{\infty}{\underset{n}{\Mapsto}}$-perform rule on ii.a and ii.c.

(e) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \overset{\infty}{\underset{\|d_1 \star d_2\|}{\Mapsto}}$ by definition $\|d_1 \star d_2\| = \|d_1\| + 1$.

(iii) $d = \texttt{let}\ d_1\ \texttt{in}\ d_2$.

Hypothesis $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}, \quad \Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \overset{\infty}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \wr \Delta$. Using the $\texttt{let}\ d_1\ \texttt{in}\ d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad c = c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet}$. We must prove $\Omega \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \overset{\infty}{\underset{\|\texttt{let}\ d_1\ \texttt{in}\ d_2\|}{\Mapsto}}$.

(a) $\Delta, s \vdash c_1 \overset{\infty}{\underset{\|d_1\|}{\Mapsto}}$ by coinduction hypothesis on $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

(b) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \overset{\infty}{\underset{\|d_1\|+1}{\Mapsto}}$ by $\overset{\infty}{\underset{n}{\Mapsto}}$-sleep on iii.a.

(c) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \overset{\infty}{\underset{\|\texttt{let}\ d_1\ \texttt{in}\ d_2\|}{\Mapsto}}$ by definition $\|\texttt{let}\ d_1\ \texttt{in}\ d_2\| = \|d_1\| + 1$.

(iv) $d = \texttt{let}\ d_1\ \texttt{in}\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow v_1, \quad v_1 \cdot \Omega \vdash d_2 \overset{\infty}{\Rightarrow}, \quad \Omega \vdash \texttt{let}\ d_1\ \texttt{in}\ d_2 \overset{\infty}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \wr \Delta$. Using the $\texttt{let}\ d_1\ \texttt{in}\ d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2,$
$c = c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet}$. We must prove $\Omega \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \overset{\infty}{\underset{\|\texttt{let}\ d_1\ \texttt{in}\ d_2\|}{\Mapsto}}$.

(a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, v_{m_1} \cdot s)$ by Theorem A2 on $\Omega \vdash d_1 \Rightarrow v_1$ (such that $v_1 \wr v_{m_1}$).

(b) $\Delta, v_{m_1} \cdot s \vdash \mathsf{ILet} \Rightarrow (v_{m_1} \cdot \Delta, s)$ by definition.

(c) $v_{m_1} \cdot \Delta, s \vdash c_2 \underset{\|d_2\|}{\overset{\infty}{\Rightarrow}}$ by coinduction hypothesis on $v_1 \cdot \Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

(d) $v_{m_1} \cdot \Delta, s \vdash c_2 \cdot \mathsf{IEndLet} \underset{\|d_2\|+1}{\overset{\infty}{\Rightarrow}}$ by $\underset{n}{\overset{\infty}{\Mapsto}}$-sleep rule on iv.c.

(e) $\Delta, v_{m_1} \cdot s \vdash \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \underset{\|d_2\|+1}{\overset{\infty}{\Rightarrow}}$ by $\underset{n}{\overset{\infty}{\Mapsto}}$-perform rule on iv.b and iv.d.

(f) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \underset{\|d_1\|+1}{\overset{\infty}{\Mapsto}}$ by $\underset{n}{\overset{\infty}{\Mapsto}}$-perform rule on iv.a and iv.e.

(g) $\Delta, s \vdash c_1 \cdot \mathsf{ILet} \cdot c_2 \cdot \mathsf{IEndLet} \underset{\|\texttt{let}\ d_1\ \texttt{in}\ d_2\|}{\overset{\infty}{\Mapsto}}$ by definition $\|\texttt{let}\ d_1\ \texttt{in}\ d_2\| = \|d_1\| + 1$.

(v) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

Hypothesis $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$, $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \overset{\infty}{\Rightarrow}$, $d \Downarrow c$, $\Omega \wr \Delta$. Using the $\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $d_3 \Downarrow c_3$, $c = c_1 \cdot \mathsf{ISel}\ c_2\ c_3$. We must prove $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\|}{\overset{\infty}{\Mapsto}}$.

(a) $\Delta, s \vdash c_1 \underset{\|d_1\|}{\overset{\infty}{\Mapsto}}$ by coinduction hypothesis on $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

(b) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|d_1\|+1}{\overset{\infty}{\Mapsto}}$ by $\underset{n}{\overset{\infty}{\Mapsto}}$-sleep rule on v.a.

(c) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\|}{\overset{\infty}{\Mapsto}}$ by definition $\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\| = \|d_1\| + 1$.

(vi) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

Hypothesis $\Omega \vdash d_1 \Rightarrow \mathit{true}$, $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$, $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \overset{\infty}{\Rightarrow}$, $d \Downarrow c$, $\Omega \wr \Delta$. Using the $\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $d_3 \Downarrow c_3$, $c = c_1 \cdot \mathsf{ISel}\ c_2\ c_3$. We must prove $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\|}{\overset{\infty}{\Mapsto}}$.

(a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, \mathit{true} \cdot s)$ by Theorem A2 on $\Omega \vdash d_1 \Rightarrow \mathit{true}$.

(b) $\Delta, s \vdash c_2 \underset{\|d_2\|}{\overset{\infty}{\Mapsto}}$ by coinduction hypothesis on $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

(c) $\Delta, \mathit{true} \cdot s \vdash \mathsf{ISel}\ c_2\ c_3 \Rightarrow \overset{\infty}{\Mapsto}$ by definition using vi.b.

(d) $\Delta, \mathit{true} \cdot s \vdash \mathsf{ISel}\ c_2\ c_3 \underset{\|d_2\|}{\overset{\infty}{\Mapsto}}$ by definition using vi.c.

(e) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|d_1\|+1}{\overset{\infty}{\Mapsto}}$ by $\underset{n}{\overset{\infty}{\Mapsto}}$-perform rule on vi.a and vi.d.

(f) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\|}{\overset{\infty}{\Mapsto}}$ by definition $\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\| = \|d_1\| + 1$.

(vii) $d = \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3$.

Hypothesis $\Omega \vdash d_1 \Rightarrow \mathit{false}$, $\Omega \vdash d_3 \overset{\infty}{\Rightarrow}$, $\Omega \vdash \texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \overset{\infty}{\Rightarrow}$, $d \Downarrow c$, $\Omega \wr \Delta$. Using the $\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $d_3 \Downarrow c_3$, $c = c_1 \cdot \mathsf{ISel}\ c_2\ c_3$. We must prove $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\|}{\overset{\infty}{\Mapsto}}$.

(a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, false \cdot s)$ by Theorem A2 on $\Omega \vdash d_1 \Rightarrow false$.

(b) $\Delta, s \vdash c_3 \underset{\|d_3\|}{\overset{\infty}{\Mapsto}}$ by coinduction hypothesis on $\Omega \vdash d_3 \overset{\infty}{\Rightarrow}$.

(c) $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_2\ c_3 \Rightarrow\ \overset{\infty}{\Mapsto}$ by definition using vii.b.

(d) $\Delta, false \cdot s \vdash \mathsf{ISel}\ c_2\ c_3 \underset{\|d_3\|}{\overset{\infty}{\Mapsto}}$ by definition using vii.c.

(e) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|d_1\|+1}{\overset{\infty}{\Mapsto}}$ by $\underset{n}{\overset{\infty}{\Mapsto}}$-perform rule on vii.a and vii.d.

(f) $\Delta, s \vdash c_1 \cdot \mathsf{ISel}\ c_2\ c_3 \underset{\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\|}{\overset{\infty}{\Mapsto}}$ by definition $\|\texttt{if}\ d_1\ \texttt{then}\ d_2\ \texttt{else}\ d_3\| = \|d_1\| + 1$.

(viii) $d = d_1\ d_2$.

Hypothesis $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}, \quad \Omega \vdash d_1\ d_2 \overset{\infty}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \wr \Delta$. Using the $d_1\ d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad c = c_1 \cdot c_2 \cdot \mathsf{IApp}$.
We must prove $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \underset{\|d_1\ d_2\|}{\overset{\infty}{\Mapsto}}$.

(a) $\Delta, s \vdash c_1 \underset{\|d_1\|}{\overset{\infty}{\Mapsto}}$ by coinduction hypothesis on $\Omega \vdash d_1 \overset{\infty}{\Rightarrow}$.

(b) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \underset{\|d_1\|+1}{\overset{\infty}{\Mapsto}}$ by $\underset{n}{\overset{\infty}{\Mapsto}}$-sleep rule on viii.a.

(c) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \underset{\|d_1\ d_2\|}{\overset{\infty}{\Mapsto}}$ by definition $\|d_1\ d_2\| = \|d_1\| + 1$.

(ix) $d = d_1\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow (\lambda.d')[\Omega_1], \quad \Omega \vdash d_2 \overset{\infty}{\Rightarrow}, \quad \Omega \vdash d_1\ d_2 \overset{\infty}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \wr \Delta$. Using the $d_1\ d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad c = c_1 \cdot c_2 \cdot \mathsf{IApp}$.
We must prove $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \underset{\|d_1\ d_2\|}{\overset{\infty}{\Mapsto}}$.

(a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, v_m \cdot s)$ by Theorem A2 on $\Omega \vdash d_1 \Rightarrow (\lambda.d')[\Omega_1]$ (such that $(\lambda.d')[\Omega_1] \mathrel{\overline{\downarrow}} v_m$).

(b) $\Delta, v_m \cdot s \vdash c_2 \underset{\|d_2\|}{\overset{\infty}{\Mapsto}}$ by coinduction hypothesis on $\Omega \vdash d_2 \overset{\infty}{\Rightarrow}$.

(c) $\Delta, v_m \cdot s \vdash c_2 \cdot \mathsf{IApp} \underset{\|d_2\|+1}{\overset{\infty}{\Mapsto}}$ by $\underset{n}{\overset{\infty}{\Mapsto}}$-sleep rule on ix.b.

(d) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \underset{\|d_1\|+1}{\overset{\infty}{\Mapsto}}$ by $\underset{n}{\overset{\infty}{\Mapsto}}$-perform rule on ix.a and ix.c.

(e) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \underset{\|d_1\ d_2\|}{\overset{\infty}{\Mapsto}}$ by definition $\|d_1\ d_2\| = \|d_1\| + 1$.

(x) $d = d_1\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d')[\Omega_1], \quad \Omega \vdash d_2 \overset{\infty}{\Rightarrow}, \quad \Omega \vdash d_1\ d_2 \overset{\infty}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \wr \Delta$. Using the $d_1\ d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1, \quad d_2 \Downarrow c_2, \quad c = c_1 \cdot c_2 \cdot \mathsf{IApp}$.
We must prove $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \underset{\|d_1\ d_2\|}{\overset{\infty}{\Mapsto}}$.

(a) $\Delta, s \vdash c_1 \Rightarrow (\Delta, v_m \cdot s)$ by Theorem A2 on $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d')[\Omega_1]$
(such that $(\mu.\lambda.d')[\Omega_1] \mathrel{\overline{\downarrow}} v_m$).

(b) $\Delta, v_m \cdot s \vdash c_2 \overset{\infty}{\Mapsto}_{\|d_2\|}$ by coinduction hypothesis on $\Omega \vdash d_2 \overset{\infty}{\Rrightarrow}$.

(c) $\Delta, v_m \cdot s \vdash c_2 \cdot \mathsf{lApp} \overset{\infty}{\Mapsto}_{\|d_2\|+1}$ by $\overset{\infty}{\Mapsto}_{n}$-sleep rule on x.b.

(d) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{lApp} \overset{\infty}{\Mapsto}_{\|d_1\|+1}$ by $\overset{\infty}{\Mapsto}_{n}$-perform rule on x.a and x.c.

(e) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{lApp} \overset{\infty}{\Mapsto}_{\|d_1\ d_2\|}$ by definition $\|d_1\ d_2\| = \|d_1\| + 1$.

(xi) $d = d_1\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow (\lambda.d')[\Omega_1]$, $\Omega \vdash d_2 \Rightarrow v_2$, $v_2 \cdot \Omega_1 \vdash d' \overset{\infty}{\Rrightarrow}$, $\Omega \vdash d_1\ d_2 \overset{\infty}{\Rrightarrow}$, $d \Downarrow c$, $\Omega \wr \Delta$. Using the $d_1\ d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot c_2 \cdot \mathsf{lApp}$. We must prove $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{lApp} \overset{\infty}{\Mapsto}_{\|d_1\ d_2\|}$.

(a) $\Omega_1 \wr \Delta_1$, $d' \Downarrow c'$, $\Delta, s \vdash c_1 \Rightarrow (\Delta, c'[\Delta_1] \cdot s)$ by Theorem A2 on $\Omega \vdash d_1 \Rightarrow (\lambda.d')[\Omega_1]$ (such that $(\lambda.d')[\Omega_1] \mapsdown c'[\Delta_1]$).

(b) $\Delta, c'[\Delta_1] \cdot s \vdash c_2 \Rightarrow (\Delta, v_{m_2} \cdot c'[\Delta_1] \cdot s)$ by Theorem A2 on $\Omega \vdash d_2 \Rightarrow v_2$ (such that $v_2 \mapsdown v_{m_2}$).

(c) $v_{m_2} \cdot \Delta_1, s \vdash c' \overset{\infty}{\Mapsto}_{\|d'\|}$ by coinduction hypothesis on $v_2 \cdot \Omega_1 \vdash d' \overset{\infty}{\Rrightarrow}$.

(d) $\Delta, v_{m_2} \cdot c'[\Delta_1] \cdot s \vdash \mathsf{lApp} \overset{\infty}{\Mapsto}$ by definition using xi.c.

(e) $\Delta, v_{m_2} \cdot c'[\Delta_1] \cdot s \vdash \mathsf{lApp} \overset{\infty}{\Mapsto}_{\|d'\|}$ by definition using xi.d.

(f) $\Delta, c'[\Delta_1] \cdot s \vdash c_2 \cdot \mathsf{lApp} \overset{\infty}{\Mapsto}_{\|d_2\|}$ by $\overset{\infty}{\Mapsto}_{n}$-perform rule on xi.b and xi.e.

(g) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{lApp} \overset{\infty}{\Mapsto}_{\|d_1\|+1}$ by $\overset{\infty}{\Mapsto}_{n}$-perform rule on xi.a and xi.f.

(h) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{lApp} \overset{\infty}{\Mapsto}_{\|d_1\ d_2\|}$ by definition $\|d_1\ d_2\| = \|d_1\| + 1$.

(xii) $d = d_1\ d_2$.

Hypothesis $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d')[\Omega_1]$, $\Omega \vdash d_2 \Rightarrow v_2$, $v_2 \cdot (\mu.\lambda.d')[\Omega_1] \cdot \Omega_1 \vdash d' \overset{\infty}{\Rrightarrow}$,
$\Omega \vdash d_1\ d_2 \overset{\infty}{\Rrightarrow}$, $d \Downarrow c$, $\Omega \wr \Delta$. Using the $d_1\ d_2 \Downarrow c$ hypothesis, by $\Downarrow$ definition necessarily $d_1 \Downarrow c_1$, $d_2 \Downarrow c_2$, $c = c_1 \cdot c_2 \cdot \mathsf{lApp}$. We must prove $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{lApp} \overset{\infty}{\Mapsto}_{\|d_1\ d_2\|}$.

(a) $\Omega_1 \wr \Delta_1$, $d' \Downarrow c'$, $\Delta, s \vdash c_1 \Rightarrow (\Delta, c'[\Delta_1]_{rec} \cdot s)$ by Theorem A2 on $\Omega \vdash d_1 \Rightarrow (\mu.\lambda.d')[\Omega_1]$
(such that $(\mu.\lambda.d')[\Omega_1] \mapsdown c'[\Delta_1]_{rec}$).

(b) $\Delta, c'[\Delta_1]_{rec} \cdot s \vdash c_2 \Rightarrow (\Delta, v_{m_2} \cdot c'[\Delta_1]_{rec} \cdot s)$ by Theorem A2 on $\Omega \vdash d_2 \Rightarrow v_2$ (such that $v_2 \mapsdown v_{m_2}$).

(c) $v_{m_2} \cdot c'[\Delta_1]_{rec} \cdot \Delta_1, s \vdash c' \overset{\infty}{\Mapsto}_{\|d'\|}$ by coinduction hypothesis on $v_2 \cdot (\mu.\lambda.d')[\Omega_1] \cdot \Omega_1 \vdash d' \overset{\infty}{\Rrightarrow}$.

(d) $\Delta, v_{m_2} \cdot c'[\Delta_1]_{rec} \cdot s \vdash \mathsf{lApp} \overset{\infty}{\Mapsto}$ by definition using xii.c.

(e) $\Delta, v_{m_2} \cdot c'[\Delta_1]_{rec} \cdot s \vdash \mathsf{IApp} \underset{\|d'\|}{\overset{\infty}{\Rightarrow}}$ by definition using xii.d.

(f) $\Delta, c'[\Delta_1]_{rec} \cdot s \vdash c_2 \cdot \mathsf{IApp} \underset{\|d_2\|}{\overset{\infty}{\Rightarrow}}$ by $\underset{n}{\overset{\infty}{\Rightarrow}}$-perform rule on xii.b and xii.e.

(g) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \underset{\|d_1\|+1}{\overset{\infty}{\Rightarrow}}$ by $\underset{n}{\overset{\infty}{\Rightarrow}}$-perform rule on xii.a and xii.f.

(h) $\Delta, s \vdash c_1 \cdot c_2 \cdot \mathsf{IApp} \underset{\|d_1\ d_2\|}{\overset{\infty}{\Rightarrow}}$ by definition $\|d_1\ d_2\| = \|d_1\| + 1$. □

Theorem A4 (Correctness for non-termination)**.** *Let Ω be a nameless environment, Δ a machine environment, d a nameless expression, c a machine code. If*

$$\Omega \vdash d \overset{\infty}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \wr \Delta$$

then, for all stack s,

$$\Delta, s \vdash c \overset{\infty}{\Rightarrow}$$

Proof. Hypothesis $\Omega \vdash d \overset{\infty}{\Rightarrow}, \quad d \Downarrow c, \quad \Omega \wr \Delta$. We must prove $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$.

(i) $\Delta, s \vdash c \underset{\|d\|}{\overset{\infty}{\Rightarrow}}$ by Lemma A5 on the hypothesis.

(ii) $\Delta, s \vdash c \overset{\infty}{\Rightarrow}$ by Lemma A4 on i. □

References

1. Leroy, X. Formal Verification of a Realistic Compiler. *Commun. ACM* **2009**, *52*, 107–115. [CrossRef]
2. Leroy, X. A formally verified compiler back-end. *J. Autom. Reason.* **2009**, *43*, 363–446. [CrossRef]
3. Leroy, X. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant. In Proceedings of the Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Charleston, SC, USA, 11–13 January 2006 2006; Association for Computing Machinery: New York, NY, USA, 2006; pp. 42–54. [CrossRef]
4. Blazy, S.; Leroy, X. Mechanized semantics for the Clight subset of the C language. *J. Autom. Reason.* **2009**, *43*, 263–288. [CrossRef]
5. Blazy, S.; Dargaye, Z.; Leroy, X. Formal Verification of a C Compiler Front-End. In *International Symposium on Formal Methods*; Misra, J., Nipkow, T., Sekerinski, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; Volume 4085, pp. 460–475. [CrossRef]
6. Anand, A.; Appel, A.W.; Morrisett, G.; Paraskevopoulou, Z.; Pollack, R.; Savary Bélanger, O.; Sozeau, M.; Weaver, M. CertiCoq: A verified compiler for Coq. In Proceedings of the The Third International Workshop on Coq for Programming Languages, CoqPL 2017, Paris, France, 15–21 January 2017.
7. Jung, R.; Krebbers, R.; Jourdan, J.H.; Bizjak, A.; Birkedal, L.; Dreyer, D. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* **2018**, *28*, e20. [CrossRef]
8. Leroy, X. Functional Programming Languages Part II: Abstract Machines. MPRI 2-4, INRIA. Available online: https://xavierleroy.org/mpri/2-4/machines.pdf (accessed on 4 March 2020).
9. Grégoire, B.; Leroy, X. A Compiled Implementation of Strong Reduction. In Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming, ICFP '02, Pittsburgh, PA, USA, 4–6 October 2002; Association for Computing Machinery: New York, NY, USA, 2002; pp. 235–246. [CrossRef]
10. Zúñiga, A.; Bel-Enguix, G. A Correct Compiler from Mini-ML to a Big-Step Machine Verified Using 1407 Natural Semantics in Coq. In Proceedings of the XVIII Jornadas de PROgramación y LEnguajes (PROLE 2018), Seville, Spain, 17–19 September 2018.
11. Kahn, G. Natural semantics. In *Annual Symposium on Theoretical Aspects of Computer Science*; Brandenburg, F.J., Vidal-Naquet, G., Wirsing, M., Eds.; Springer: Berlin/Heidelberg, Germany, 1987; Volume 247, pp. 22–39. [CrossRef]

12. Despeyroux, J. Proof of Translation in Natural Semantics. Research Report RR-0514; INRIA. 1986. Available online: https://hal.inria.fr/inria-00076040/file/RR-0514.pdf (accessed on 4 March 2020).
13. Boutin, S. Proving Correctness of the Translation from Mini-ML to the CAM with the Coq Proof Development System. Research Report RR-2536; INRIA. Available online: https://hal.inria.fr/inria-00074142/file/RR-2536.pdf (accessed on 4 March 2020).
14. Leroy, X. Coinductive Big-Step Operational Semantics. In Proceedings of the ESOP 2006: Programming Languages and Systems, Vienna, Austria, 27–28 March 2006; Sestoft, P., Ed.; Springer: Berlin/Heidelberg, Germany, 2006; Volume 3924, pp. 54–68. [CrossRef]
15. Leroy, X.; Grall, H. Coinductive big-step operational semantics. *Inf. Comput.* **2009**, *207*, 284–304. [CrossRef]
16. Hardin, T.; Maranget, L.; Pagano, B. Functional runtime systems within the lambda-sigma calculus. *J. Funct. Program.* **1998**, *8*, 131–176. [CrossRef]
17. Grégoire, B. Compilation des Termes de Preuves: Un (nouveau) Mariage Entre Coq et OCaml. Ph.D. Thesis, Spécialité Informatique, Université Paris 7—Denis Diderot, École Polytechnique, France, 2003.
18. Leroy, X. The ZINC Experiment: An Economical Implementation of the ML Language. Technical Report RT-0117; INRIA. Available online: https://hal.inria.fr/inria-00070049/file/RT-0117.pdf (accessed on 4 March 2020).
19. Kunze, F.; Smolka, G.; Forster, Y. Formal Small-Step Verification of a Call-by-Value Lambda Calculus Machine. In Proceedings of the Asian Symposium on Programming Languages and Systems, APLAS 2018, Wellington, New Zealand, 2–6 December 2018; Ryu, S., Ed.; Springer International Publishing: Cham, Switzerland, 2018; Volume 11275, pp. 264–283. [CrossRef]
20. Charguéraud, A. Pretty-Big-Step Semantics. In Proceedings of the ESOP 2013: Programming Languages and Systems, Rome, Italy, 16–24 March 2013; Felleisen, M., Gardner, P., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7792, pp. 41–60. [CrossRef]
21. Bach Poulsen, C.; Mosses, P.D. Flag-based big-step semantics. *J. Log. Algebr. Methods Program.* **2017**, *88*, 174–190. [CrossRef]
22. Leroy, X. Le Système Caml Special Light: Modules et Compilation Efficace en Caml. Research Report RR-2721; INRIA. Available online: https://hal.inria.fr/inria-00073972/file/RR-2721.pdf (accessed on 4 March 2020).
23. Dargaye, Z. Vérification Formelle d'un Compilateur Optimisant Pour Langages Fonctionnels. Ph.D. Thesis, Université Paris 7—Denis Diderot, Paris, France, 2009.
24. Sozeau, M.; Anand, A.; Boulier, S.; Cohen, C.; Forster, Y.; Kunze, F.; Malecha, G.; Tabareau, N.; Winterhalter, T. The MetaCoq Project. *J. Autom. Reason.* **2020**, *64*, 947–999. [CrossRef]
25. Sozeau, M.; Boulier, S.; Forster, Y.; Tabareau, N.; Winterhalter, T. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proc. ACM Program. Lang.* **2019**, *4*, 1–28. [CrossRef]
26. Barras, B. Auto-Validation d'un Système de Preuves Avec Familles Inductives. Ph.D. Thesis, Université Paris 7—Denis Diderot, Paris, France, 1999.
27. Ager, M.S. From Natural Semantics to Abstract Machines. In Proceedings of the LOPSTR 2004: Logic Based Program Synthesis and Transformation, London, UK, 7–9 September 2005; Etalle, S., Ed.; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3573, pp. 245–261. [CrossRef]
28. Biernacka, M.; Danvy, O. A Concrete Framework for Environment Machines. *ACM Trans. Comput. Log.* **2007**, *9*, 6-es. [CrossRef]
29. Biernacka, M.; Biernacki, D. Formalizing Constructions of Abstract Machines for Functional Languages in Coq. In Proceedings of the 7th International Workshop on Reduction Strategies in Rewriting and Programming, Paris, France, 25 June 2007; Giesl, J., Ed.; Preeliminary Proceedings: Paris, France, 2007; pp. 84–99.
30. Pirog, M.; Biernacki, D. A Systematic Derivation of the STG Machine Verified in Coq. In Proceedings of the third ACM Haskell symposium on Haskell, Baltimore, MA, USA, 30 September 2010; Association for Computing Machinery: New York, NY, USA, 2010; Haskell '10, pp. 25–36. [CrossRef]
31. Sieczkowski, F.; Biernacka, M.; Biernacki, D. Automating Derivations of Abstract Machines from Reduction Semantics. In Proceedings of the IFL 2010: Implementation and Application of Functional Languages, Alphen aan den Rijn, The Netherlands, 1–3 September 2010; Hage, J., Morazán, M.T., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6647, *LNCS*, pp. 72–88. [CrossRef]

32. Biernacka, M.; Charatonik, W.; Zielinska, K. Generalized Refocusing: From Hybrid Strategies to Abstract Machines. In Proceedings of the 2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017), Oxford, UK, 3–9 September 2017; Miller, D., Ed.; Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2017; Volume 84, pp. 10:1–10:17. [CrossRef]
33. Biernacka, M.; Charatonik, W. Deriving an Abstract Machine for Strong Call by Need. In Proceedings of the 4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019), Dortmund, Germany, 24–30 June 2019; Geuvers, H., Ed.; Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2019; Volume 131, pp. 8:1–8:20. [CrossRef]
34. Danvy, O.; Nielsen, L.R. *Refocusing in Reduction Semantics*. Technical Report RS-04-26, BRICS; Department of Computer Science, University of Aarhus: Aarhus, Denmark, 2004.
35. Chlipala, A. A Verified Compiler for an Impure Functional Language. In Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Madrid, Spain, 20–22 January 2010; Association for Computing Machinery: New York, NY, USA, 2010; pp. 93–106. [CrossRef]
36. Benton, N.; Hur, C.K. Biorthogonality, Step-Indexing and Compiler Correctness. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, Edinburgh, Scotland, 31 August–2 September 2009; Association for Computing Machinery: New York, NY, USA, 2009; pp. 97–108. [CrossRef]
37. Werner, B. Une Théorie des Constructions Inductives. Ph.D. Thesis, Université Paris-Diderot—Paris VII, Paris, France, 1994.
38. Glondu, S. Towards Certification of the Extraction of Coq. Ph.D. Thesis, Université Paris 7—Denis Diderot, Paris, France, 2012.
39. Mullen, E.; Pernsteiner, S.; Wilcox, J.R.; Tatlock, Z.; Grossman, D. CEuf: Minimizing the Coq Extraction TCB. In Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, Los Angeles, CA, USA, 8–9 January 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 172–185. [CrossRef]
40. Savary Bélanger, O. Verified Extraction for Coq. Ph.D. Thesis, Princeton University, Princeton, NJ, USA, 2019.
41. Bélanger, O.S.; Appel, A.W. Shrink Fast Correctly! In Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, 9–10 October 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 49–60. [CrossRef]
42. Paraskevopoulou, Z.; Appel, A.W. Closure Conversion is Safe for Space. *Proc. ACM Program. Lang.* **2019**, *3*, 1–29. [CrossRef]
43. Ahmed, A. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In Proceedings of the ESOP 2006: Programming Languages and Systems, Vienna, Austria, 27–28 March 2006; Sestoft, P., Ed.; Springer: Berlin/Heidelberg, Germany, 2006; Volume 3924, pp. 69–83. [CrossRef]
44. Ahmed, A.; Blume, M. Typed Closure Conversion Preserves Observational Equivalence. In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, Victoria, BC, Canada, 22–24 September 2008; Association for Computing Machinery: New York, NY, USA, 2008; pp. 157–168. [CrossRef]
45. Patrignani, M.; Ahmed, A.; Clarke, D. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* **2019**, *51*, 1–36. [CrossRef]
46. Abate, C.; Blanco, R.; Ciobâcă, S.; Durier, A.; Garg, D.; Hritcu, C.; Patrignani, M.; Tanter, É.; Thibault, J. Trace-Relating Compiler Correctness and Secure Compilation. In Proceedings of the ESOP 2020: Programming Languages and Systems, Dublin, Ireland, 27–30 April 2020; Müller, P., Ed.; Springer International Publishing: Cham, 2020; Volume 12075, pp. 1–28. [CrossRef]
47. Perconti, J.T.; Ahmed, A. Verifying an Open Compiler Using Multi-language Semantics. In Proceedings of the ESOP 2014: Programming Languages and Systems, Grenoble, France, 5–13 April 2014; Shao, Z., Ed.; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8410, pp. 128–148. [CrossRef]
48. Neis, G.; Hur, C.K.; Kaiser, J.O.; McLaughlin, C.; Dreyer, D.; Vafeiadis, V. Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language. In Proceedings of the ICFP 2015, The 20th ACM SIGPLAN International Conference on Functional Programming, Vancouver, BC, Canada, 31 August–2 September 2015; Association for Computing Machinery: New York, NY, USA, 2015; pp. 166–178. [CrossRef]

49. Patterson, D.; Ahmed, A. The next 700 Compiler Correctness Theorems (Functional Pearl). *Proc. ACM Program. Lang.* **2019**, *3*, 1–29. [CrossRef]
50. Dreyer, D.; Ahmed, A.; Birkedal, L. Logical Step-Indexed Logical Relations. *Log. Methods Comput. Sci.* **2011**, *7*. [CrossRef]
51. Jung, R.; Swasey, D.; Sieczkowski, F.; Svendsen, K.; Turon, A.; Birkedal, L.; Dreyer, D. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Mumbai, India, 15–17 January 2015; Association for Computing Machinery: New York, NY, USA, 2015; pp. 637–650. [CrossRef]
52. Jung, R.; Krebbers, R.; Birkedal, L.; Dreyer, D. Higher-Order Ghost State. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, Nara, Japan, 18 September 2016; Association for Computing Machinery: New York, NY, USA, 2016; ICFP 2016, pp. 256–269. [CrossRef]
53. Krebbers, R.; Jung, R.; Bizjak, A.; Jourdan, J.H.; Dreyer, D.; Birkedal, L. The Essence of Higher-Order Concurrent Separation Logic. In Proceedings of the ESOP 2017: Programming Languages and Systems, Uppsala, Sweden, 25–28 April 2017; Yang, H., Ed.; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10201, pp. 696–723. [CrossRef]
54. Linn Georges, A.; Trieu, A.; Birkedal, L. Mechanized Reasoning about a Capability Machine. Available online: https://iris-project.org/pdfs/2020-iris-capabilities-prisc-conf.pdf (accessed on 4 March 2020).
55. Cuellar, S.; Giannarakis, N.; Madiot, J.M.; Mansky, W.; Beringer, L.; Cao, Q.; Appel, A.W. *Compiler Correctness for Concurrency: From Concurrent Separation Logic to Shared-Memory Assembly Language*. Technical Report TR-014-19; Department of Computer Science, Princeton University: Princeton, NJ, USA, 2020.
56. Cuellar, S. Concurrent Permission Machine for Modular Proofs of Optimizing Compilers with Shared Memory Concurrency. Ph.D. Thesis, Princeton University, Princeton, NJ, USA, 2020.
57. Nakata, K.; Uustalu, T. Trace-Based Coinductive Operational Semantics for While. In Proceedings of the International Conference on Theorem Proving in Higher Order Logics 2009, Munich, Germany, 17–20 August 2009; Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; Volume 5674, pp. 375–390. [CrossRef]
58. Leroy, X. L'éternité c'est Long … Sémantiques de la Divergence: Théorie des Domaines et Approches Coinductives. Sémantiques Mécanisées, Sixième Cours, Collège de France. 2020. Available online: https://www.college-de-france.fr/media/xavier-leroy/UPL2331162062302586657_leroy_cours_6.pdf (accessed on 4 March 2020).
59. Leroy, X. Personal communication, 2020.
60. Paulin-Mohring, C. Extraction de Programmes dans le Calcul des Constructions. Ph.D. Thesis, Université Paris-Diderot—Paris VII, Paris, France, 1989.
61. Zúñiga, A. (Coinductive) Natural Semantics for Compiler Verification in Coq: The Coq Development. Available online: https://sites.google.com/a/ciencias.unam.mx/zuniga/repository/cnsvcompiler.tgz (accessed on 24 August 2020).
62. Jourdan, J.H.; Laporte, V.; Blazy, S.; Leroy, X.; Pichardie, D. A Formally-Verified C Static Analyzer. *ACM SIGPLAN Not.* **2015**, *50*, 247–259. [CrossRef]
63. Jourdan, J.H.; Pottier, F.; Leroy, X. Validating LR(1) Parsers. In Proceedings of the European Symposium on Programming, ESOP 2012, Tallinn, Estonia, 24 March–1 April, 2012; Seidl, H., Ed.; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7211, pp. 397–416. [CrossRef]
64. Landin, P.J. The Mechanical Evaluation of Expressions. *Comput. J.* **1964**, *6*, 308–320. [CrossRef]
65. Henderson, P. *Functional Programming Application and Implementation*; Computer Science, Prentice-Hall International: Upper Saddle River, NJ, USA, 1980.
66. Giménez, E. Un Calcul de Constructions Infinies et Son Application a la Vérification de Systemes Communicants. Ph.D. Thesis, École Normale Supérieure de Lyon, Lyon, France, 1996.
67. Tollitte, P.N.; Delahaye, D.; Dubois, C. Producing Certified Functional Code from Inductive Specifications. In Proceedings of the International Conference on Certified Programs and Proofs, Kyoto, Japan, 13–15 December 2012; Hawblitzel, C., Miller, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7679, pp. 76–91, CPP 2012. [CrossRef]
68. Dubois, C.; Ménissier-Morain, V. Certification of a Type Inference Tool for ML: Damas–Milner within Coq. *J. Autom. Reason.* **1999**, *23*, 319–346. [CrossRef]

69. Pereira, F.C.N.; Warren, D.H.D. Parsing as Deduction. In Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, ACL '83, Cambridge, MA, USA, 15–17 June 1983; Association for Computational Linguistics: Cambridge, MA, USA, 1983; pp. 137–144. [CrossRef]
70. Shieber, S.M.; Schabes, Y.; Pereira, F.C.N. Principles and implementation of deductive parsing. *J. Log. Program.* **1995**, *24*, 3–36. [CrossRef]