Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Automated Derivation of Abstract Machines from Reduction Semantics:

A Formalisation of the Refocusing Transformation in the Coq Proof System

Filip Sieczkowski

A Master's Thesis written under the supervision of
Dariusz Biernacki

Wrocław, June 2010

Abstract

In this thesis we present a formalisation of the refocusing transformation, introduced by Danvy and Nielsen, in the Coq proof assistant. Refocusing is a well known and widely used transformation, allowing to obtain an abstract machine starting with reduction semantics of a language. However, it has only been used as an informal procedure, and has never been adequately formalised.

We propose an axiomatisation of refocusing, and provide a proof that any language conforming to the axiomatisation can be transformed to obtain an abstract machine. Due to the use of the Coq proof assistant, this proof also provides a framework for automatic derivation of abstract machines from reduction semantics. We also present several case studies of different complexity, to illustrate the usage of the framework.

Aside from the basic transformation, we also formalise and prove three extensions of refocusing. Two of those have already been presented in the literature – though, as in the case of the base transformation, they were not formalised. The third extension, allowing to reason about the potentially nonterminating programs, has not been proposed before. For each of these extensions we provide case studies both in the thesis and in the accompanying Coq development.[1]

[1]The Coq developments accompanying this thesis can be found at the author's web page, `http://fsieczkowski.com`.

Acknowledgements

I would like to thank my supervisor, Dariusz Biernacki, for introducing me to this project, and his guidance and advice during the year of my work on it. He has dedicated great amount of his time and patience to me during this time, and I am most grateful for it. I would also like to thank Małgorzata Biernacka for reading the draft versions of this thesis; her corrections and kind words were invaluable.

Special thanks goes to my friend and fellow student Maciej Piróg, for fun times exchanging and refining the ideas that came to form our theses and for the long days of writing them up. He has also helped me with many small problems I have encountered when I was beginning to learn Coq, which greatly sped up my work. I should also mention all the staff and students of the Institute of Computer Science of the University of Wrocław. The four years I have spent here have taught me much more then I have ever thought possible. Last, but not least, I want to thank all my friends, who kept me sane during the last few weeks. Without them I wouldn't have got by.

Wrocław, July 2010

Filip Sieczkowski

Contents

Chapter 1

Introduction

Since the beginning of formal studies of programming languages there has been much interest in equivalence between different semantics of a language. Of special interest is the correctness of semantics used in the process of compilation of the language, the most common being abstract machines. They are generally less intuitive and more involved then natural semantics [22] or structured operational semantics [31]. Some of the first examples of such machines were Landin's SECD [24], proved correct by Plotkin [30] and Krivine's machine, introduced in [23] and proved correct by Crégut[9].

Before we delve into interderivations of semantics, we turn to the taxonomy of abstract machines. Although they have been used in programming language semantics almost as long as the field exists, until recently there was no general consensus on what an abstract machine really is. It was generally understood that abstract machine is a state transition system, with transitions mapping one *configuration* into another[29]. The configuration can consist of several elements, depending on the machine, one or more of which directs the machine's transitions. However, it was not clear what should these elements be, and the term virtual machine has also sometimes also been used to denote similar transition systems. Ager et al. have suggested in [2] to distinguish between the two depending on the existence of an instruction set: a virtual machine has one, while an abstract machine does not, working directly on the terms of the language. This contrasts with some machines existing earlier, for example the Categorical Abstract Machine (CAM)[8], which by this definition is actually a virtual machine. However, the definition gives a clear and rather fitting distinction between the two, and seems to become part of the folklore.

There exist two main subdivisions of abstract machines, based on their treatment of arguments [27].[1] In an eval-apply machine, the *caller*, while calling a function has to pass it the right number of arguments. This is realised by checking the number of arguments on the stack at runtime, and calling the appropriate version of the function (possibly creating a partial application). In a push-enter machine, on the other hand, the function called is required to check the number of arguments supplied and do the appropriate computation. The push-enter model seems intuitively more efficient, which has been one of the reason for the popularity of right-to-left evaluation order with the implementers of eager functional languages[25]. However, this has turned out not to be true, as Marlow and Peyton Jones have shown in [27],

[1] The distinction also applies to the virtual machines, in an analogous way.

due to a big overhead caused by the stack inspection in further stages of compilation. Nonetheless, push-enter abstract machines are also interesting, at least from semanticist's point of view.

Although, as we have mentioned, the attempts to prove the correctness of abstract machines have been made since the moment they were introduced, the first time we can say the machine has actually been *derived* – starting from a simple evaluator, and following a sequence of steps to obtain it – was Felleisen and Flatt's reconstruction of the CEK machine [16]. The machine in question, however has been already known and the derivation has been specifically tailored to fit it. A big step in the direction of general transformations of semantics was Reynolds' paper [32], where the derivations presented as informal, yet generic procedures, are used to transform evaluators into a continuation passing style and eliminate the higher order functions via defunctionalisation into abstract machine-based evaluators – and the other way round. This way, new abstract machines could be designed by simply following these transformations. This insight has given rise to studies on more formal aspect of both these procedures, as well as other transformations of evaluators of programming languages[2, 13, 1].

One of such transformations is refocusing, introduced by Danvy and Nielsen in [13] and later extended by Biernacka and Danvy[6, 5]. This transformation, unlike the ones introduced by Reynolds or by Ager in [1], which use natural semantics, starts with a reduction semantics of a language. This formalism, also known as *syntactic theory*, is a variant of small-step operational semantics that uses explicit evaluation contexts[15, 16]. It is a popular framework, especially should one need to reason about evaluation contexts which in the classic operational semantics are present only implicitly, as the structure of the proof tree in SOS, and even more hidden in big-step semantics. We will provide more details on the reduction semantics in Chapter 2, as one of the main parts of this thesis is its axiomatisation.

The refocusing transformation as described in [13] starts with an evaluator of reduction semantics, and proceeds with a series of small changes to the evaluator to arrive at an abstract machine. The procedure does not require insight other then the structure of the evaluator to follow, so it can be done almost automatically. It has been used, in many variations, to obtain abstract machines already well known [13, 6], as well as some modifications of such machines, mainly having to do with control operators[5], but also extending to more sophisticated languages, like a subset of Scheme[3]. The most poignant application of this transformation in recent years has been its role in obtaining abstract machine for call-by-need evaluation. In [18], Garcia et al., while not using refocusing directly, acknowledge being inspired by the transformation. Danvy et al., on the other hand, manage to apply the whole transformation to the slightly transformed calculus, and derive a machine much simpler then the previous one[12]. In both these cases, a new machine, working in a novel way is obtained much more easily then if one had to design it from scratch – which is very important when one considers sophisticated languages. All these results highlight the importance of studying such transformations and proving them in general case, rather then trying to conjure an abstract machine using only one's intuition and underlines the strength of refocusing.

1.1 Related work

As we have mentioned before, refocusing is not the only approach to transform semantics into abstract machines. The historically first, and very popular transformation is the CPS transformation combined with Reynolds' defunctionalisation, which was already mentioned. This transformation was used to derive numerous implementations of abstract machines, for example by Ager et al.[2]. In the same paper, they have also used the converse transformation – refunctionalisation and transformation to direct style – to arrive at an evaluator starting from an abstract machine. This converse transformation requires much more insight then going from evaluator to a machine, but can be successfully applied if, for example, one wishes to propagate changes made to the machine back to the evaluator.

In [1], Ager presented a transformation working directly on the rules of a special class of natural semantics, obtaining abstract machine-based semantics of a language. This was a very important notion, as it moved the transformations from the level of evaluators – implementing a semantics, but written in some other language – to the level of semantics itself. The technique used is similar in spirit to the earlier transformations of evaluators, as well as to the one employed by Diehl in his natural semantics-directed construction of abstract machines[14]. Somewhat similar in spirit is also the technique employed by Hannan and Miller in [19], although they begin with a very specific kind of small-step operational semantics heavily influenced by proof theory.

Another approach to interderivations of semantics is presented by Hardin et al. in [20]. They decompile the virtual machine's instructions or abstract machine's terms into a specific calculus of explicit substitutions, $\lambda\rho_w$, which allows them to reason about the reduction strategy implemented by the machine. This approach also lets them prove not only the extensional equality of evaluation functions, but also a guarantee that each transition of a machine is simulated by a series of steps in their calculus. This result relates in a way to the result we show in Chapter 8 where we also relate the reduction semantics to obtained abstract machine more intensionally then just with respect to the evaluation function.

1.2 Goals and approach

As the examples presented before illustrate, refocusing is now a well known transformation, and becomes used more and more widely. However, since the seminal paper by Danvy and Nielsen [13], there has been little interest in the foundations of this derivation and its axiomatisation has never been explicitly given. This state of things has been one of the motivations for undertaking the work on formalising the transformation; the second one was to design a framework that could be used to automatically apply the transformation and obtain an abstract machine from reduction semantics.

These goals have found an excellent means of realisation in the Coq proof assistant, a tool which can do the tedious work of checking the correctness of a long, complicated proof. It also features a purely functional programming language powerful enough to conveniently express a concrete reduction semantics and plug it into a generic transformer to get a new, more efficient semantics – along with a proof of its equivalence to the one at the starting point. We build our approach upon preliminary results by Biernacka and Biernacki[4], where they have used the Coq proof

assistant to derive abstract machines from reduction semantics for two case studies.

Our approach consists of two parts: first is to specify a signature – or a 'module type' in Coq nomenclature – that consists of definitions and axioms of a reduction semantics that can undergo the refocusing transformation, and second – to create a series of functors that would correspond to different stages of the refocusing transformation and guarantee the equivalence of input and output semantics. We implement such formalisation for the basic refocusing transformation, as well as three extensions: one that lets us create more efficient abstract machines that use explicit environments, one that allows to transform languages that use context-sensitive reduction (e.g., languages with multiple binders or control operators), both due to Biernacka and Danvy[6], and one that makes it possible to reason about non-terminating programs. These extensions are orthogonal to each other – any number of them may be used at one time.

1.3 Overview

The rest of the thesis is structured as follows. In Chapter 2 we axiomatise the substitution based reduction semantics and a version of reduction semantics for which automatic refocusing is possible. In Chapter 3 we follow the refocusing method starting with semantics from Chapter 2 and obtain abstract machine semantics in two versions: an eval-apply abstract machine and – with additional requirements on semantics – a push-enter machine. In Chapter 4 we show three case studies of varying complexity illustrating the use of the general derivation shown in Chapter 3. In Chapter 5 we elaborate on the implementation of the derivation in the Coq interactive theorem prover. The next three chapters are reserved for extensions of refocusing transformation. In Chapter 6 we describe the reduction semantics based on Curien's calculus of closures refined by Biernacka and Danvy in [6] and show how the use of this formalism allows to automatically obtain abstract machines with environments, along with case studies. In Chapter 7 we extend the semantics with the notion of context-sensitive reduction following [6] and [5], and use it to obtain abstract machines for languages with control operators and multiple binders. In Chapter 8, finally, we extend the semantics to potentially non-terminating programs and show the correctness of refocusing also for this case. We finish the thesis with discussion of conclusions and perspectives in Chapter 9.

Chapter 2

An axiomatisation of a substitution-based reduction semantics

In this chapter we describe the axiomatisation of a generic reduction semantics and its extension to semantics that can be automatically refocused. The description is similar to the one given by Danvy and Nielsen in [13], though it does differ in some points – most notably we require potential redexes to be explicitly provided, whereas Danvy and Nielsen specify them by their properties. This and similar differences are used to better fit the implementation in the proof assistant. As an example we present how a language of simple arithmetic expression fulfils this specification.

2.1 Syntactic categories

We begin by specifying the syntactic categories used throughout the formalisation: terms, values, potential redexes and context frames, which we denote with t, v, r and f respectively. Both values and potential redexes should be subsets of terms; moreover, the sets of values and redexes should be disjoint. Potential redexes describe terms, which can possibly contract, reducing into a new term. However, we do not require contraction of every potential redex to be successful. We elaborate the reasons for such specification in the next section, where we discuss contraction in more detail.

The syntactic category of reduction contexts (denoted as E) is defined as lists of frames and interpreted similarly to classic inside-out reduction contexts (that is, as a stack). Composition of two contexts is denoted as $E_1 \circ E_2$, while a context extended with a single frame – as $f :: E$.

The meaning of reduction contexts is usually specified by a *plug* function, which describes the effect of putting a term in the context. As the reduction contexts are, in our axiomatisation, secondary constructs, we specify *plug* as a (left) folding of an *atomic plug* function over reduction context, where *atomic plug* describes the effect of plugging a term into a context frame. We will denote plugging of a term t into a context E with $E[t]$. Our approach enforces that composition of contexts and *plug* satisfy $(E_1 \circ E_2)[t] = E_2[E_1[t]]$, which otherwise would have to be proved.

We use a simple language of arithmetic expressions with constants and addition,

denoted with $\oplus$, as an example throughout this chapter. Its syntactic categories can be defined by the following grammars, where n ranges over natural numbers:

$$
\begin{aligned}
t &::= n \mid t \oplus t && \text{(terms)}\\
v &::= n && \text{(values)}\\
r &::= v \oplus v && \text{(redexes)}\\
f &::= [\,] \oplus t \mid v \oplus [\,] && \text{(context frames)}
\end{aligned}
$$

The atomic plug function is defined in obvious way as replacing the hole in the frame by the term plugged.

2.2 Decompositions and contraction

The notion of decomposition is also defined as in [13] – any pair (E, t) is a decomposition of $E[t]$. A decomposition of the form (E, v), where v is a value is said to be *trivial*, while the decomposition $([\,], t)$ is called *empty*. Our axiomatisation requires that for both values and potential redexes every nonempty decomposition is trivial and that every term that has only trivial or empty decompositions is either a value or a potential redex. Intuitively, this means that values are normal forms – and as such, if they can be decomposed, the subterm should also be a value – while potential redexes are minimal non-value terms. We also need a partial function *contract* that takes a potential redex, and returns a term resulting from reducing it. Of course, not every potential redexes can actually be contracted – we say the term is *stuck* if such thing happens.

It is evident that this formalisation might preclude treating all weak head normal forms as values, as a potential redex that cannot be contracted is also a normal form. However, in programming languages, the stuck term generally represents a program that "goes wrong," as the actual values can in most cases be easily specified and a stuck term can never be a value. A following example is a good illustration of how the term becomes stuck because of program going wrong. Let us consider a programming language with a pair constructor and projections, where a contraction rule requires that the projection be applied to the pair built with the constructor. In such a language, the term goes stuck if the projection is applied to a value other then a pair – a situation that in any programming language would be reported as an error.

Let us get back to the arithmetic expressions example. We notice, that values can never be decomposed into a term and a nonempty context. It follows that redexes can only be decomposed trivially or into an empty context and the redex itself – any redex $r = v \oplus v'$ can be either decomposed as $([\,], r)$, $([\,] \oplus v', v)$ or $(v \oplus [\,], v')$, so the obligations on decompositions of values and redexes are fulfilled. The requirement that a term with only trivial or empty decompositions is either a value or a redex is simple to prove: take term t. The only way that t could not be a value is if it were an addition – so t has to take the form of $t_1 \oplus t_2$. Obviously, it can be decomposed as $([\,] \oplus t_2, t_1)$. All the decompositions of t are trivial, though, so t_1 is a value – call it v_1. However, now we know that t can also be decomposed as $(v_1 \oplus [\,], t_2)$, but this decomposition also has to be trivial, so $t_2 = v_2$ for some value v_2 – and we notice that t is actually a potential redex.

Contraction in our example is defined as actually performing the addition, with the rule being: $n \oplus m \mapsto n + m$. Of course, with such definition no stuck terms can appear. Both the semantics before and after the refocusing transformation are defined using the contraction as a black box, and for the basic transformation we do not require any specific properties of this function.

2.3 Reduction semantics

For a language satisfying the conditions mentioned above we can specify a reduction semantics. This can be done in a few ways, which we will go through in the following paragraphs. First, we will specify the basic properties that any deterministic reduction semantics has to possess. Then, we will add some rules that make a semantics suitable for automatic refocusing transformation. Lastly, we will provide a different set of axioms that is much independent of the more sophisticated properties of the semantics, yet still yields a definition that complies with the automatic refocusing rules.

2.3.1 Required properties

To specify a deterministic reduction semantics, we need a *dec* function, which finds a decomposition of a given term in a reduction context. It is specified as a relation to avoid problems with having to prove its termination. We require the user to prove the following properties, which state that the decomposition is compatible with plugging and deterministic.

$$\forall t.\, \exists v.\, t = v \vee \exists r\, E.\, t = E[r] \qquad \text{(decompose)}$$

$$\texttt{dec}\, t\, E\, d \implies E[t] = \begin{cases} v & \text{if } d = v \\ E'[r] & \text{if } d = (E', r) \end{cases} \qquad \text{(decompose-correct)}$$

$$\texttt{dec}\, t\, (E \circ E')\, d \iff \texttt{decompose}(E[t])\, E'\, d \qquad \text{(decompose-plug)}$$

$$\texttt{dec}\, r\, E\, (r, E) \qquad \text{(decompose-redex)}$$

These properties allow us to prove that `decompose` is a total function and that for any non-value term there exists a unique decomposition of that term into a potential redex and a reduction context. The first property states the requirement on the compatibility, the second allows for skipping large parts of the "spine" of the term being decomposed, while the third one allows to reason about decomposition of a redex without taking into account what is the structure of values that are its subterms. All these obligations are natural for any deterministic reduction semantics – and they allow to prove some more properties that are desired of reduction semantics, especially the unique decomposition property.

We can obviously create a generic wrapper that would execute the contract-plug-decompose loop until the decomposition returns a value, as well as an evaluation function realising the specified semantics. These functions are, of course, potentially non-terminating, so they are also specified as relations.

We will refrain from giving a decomposition function for our semantics here and only do it for the automatic refocusing-enabled semantics.

2.3.2 Reduction semantics allowing for automatic refocusing

The preceding conditions are not sufficient for automatic application of the refocusing transformation. It consists for the most part of changes in the decomposition relation and the proofs are in most cases done by induction on the structure of decomposition, a method which we cannot apply relying only on this axiomatisation. To deal with this problem, we notice that the decomposition of a term t can lead to one of three possibilities:

- t is a redex r, that cannot be further decomposed
- t is a value v, that also cannot be decomposed (e.g. a lambda form or lazy constructor)
- t can be decomposed into a term t' and a context frame f

The first and third cases are straightforward – either we have found a decomposition by locating a potential redex, or we can decompose t' in a context enhanced with f. In the second case, we have to look at the current context: if it is empty, we have also finished the decomposition. Otherwise, we can examine the innermost context frame f and the value v. They can either form a potential redex or a value – when all the decompositions of $f[v]$ have been checked as trivial, or a term t' and context frame f', that has a hole in a different place then f. We require the user to provide two functions capturing this insight: $\mathtt{dec}_t$, that for a given term describes how it can be decomposed in one step, and an analogous function $\mathtt{dec}_f$, that does this for a context frame and a value. These "atomic" decomposition functions let us specify the shape of the decomposition predicate in terms of them. Of course now we also need the proof that these user-defined functions are correct with respect to the (atomic) plug function, as well as the properties from the previous specification. The decomposition relation can now iterate the execution of required functions until a decomposition is found, taking the following form:

$$\mathtt{dec}\, t\, E\, d \iff \begin{cases} d = (r, E), & \text{if } \mathtt{dec}_t\, t = r \\ \mathtt{dec}_E\, E\, v\, d, & \text{if } \mathtt{dec}_t\, t = v \\ \mathtt{dec}\, t'\, (f :: E)\, d, & \text{if } \mathtt{dec}_t\, t = (t', f) \end{cases}$$

$$\mathtt{dec}_E\, [\,]\, v\, d \iff d = v$$

$$\mathtt{dec}_E\, (f :: E)\, v\, d \iff \begin{cases} d = (r, E), & \text{if } \mathtt{dec}_f\, f\, v = r \\ \mathtt{dec}_E\, E\, v'\, d, & \text{if } \mathtt{dec}_f\, f\, v = v' \\ \mathtt{dec}\, t\, (f' :: E)\, d & \text{if } \mathtt{dec}_f\, f\, v = (t, f') \end{cases}$$

In the arithmetic expressions example, we can define the "atomic" decomposition functions as:

$$\begin{aligned} \mathtt{dec}_t\, n &= n \\ \mathtt{dec}_t\, (t \oplus t') &= (t, [\,] \oplus t') \\ \mathtt{dec}_f\, ([\,] \oplus t)\, v &= (t, v \oplus [\,]) \\ \mathtt{dec}_f\, (v \oplus [\,])\, v' &= v \oplus v' \end{aligned}$$

We can observe that these functions are compatible with the atomic plugging operation; with some more work we could also show the remaining properties of Section 2.3.1. We will not show those properties, though, as in the next section we will see that utilising a slightly different approach we need not do it at all.

2.3.3 A strict specification of reduction semantics

The last – and most interesting – axiomatisation constructs the reduction semantics, along with needed proofs, automatically. As a tradeoff, there are greater requirements on the language, though, in all the case studies, these obligations are much easier to prove then the ones resulting from following the specification of Section 2.3.2. As previously, the user is required to provide the atomic decomposition functions, along with the proofs of their correctness with respect to atomic plug. These are, however, easy proofs by case, and the only ones having to do with semantics of the language – and, apart from contraction operation, they contain the whole semantics of the language.

The major part of this axiomatisation are two orderings: a subterm order $\prec_t$ which states that one term is a subterm of the other, and an order on context frames $\prec_f$ that describes the order of evaluation of subterms. In the arithmetic expressions example, it would be defined as a smallest strict order in which $v \oplus [\,] \prec_f [\,] \oplus t$ for any term t and value v. This should hold for any t and v, because the term $(v \oplus t)$ can be decomposed into both contexts. The subterm order is specified using the atomic plug function, and the only property we require is its well-foundedness, which in most cases can be proved by easy induction: this property carries the information that the terms of the language are actually terms (or at least can be viewed as ones). The order on context frames, on the other hand, is left to be defined by the user – we only require that several properties (beside it being well-founded) hold. These properties state that the order describes the same order of evaluation of subterms that the atomic decomposition functions describe, that is:

- if $\mathtt{dec}_t\, t = (t', f)$, then f is maximal with respect to $\prec_f$
- if $\mathtt{dec}_f\, f\, v = (t, f')$, then $f' \prec_f f$ and $\forall f''.f'' \prec_f f \implies f'' \preceq_f f'$
- if $\mathtt{dec}_f\, f\, v$ gives a redex or a value, then f is minimal with respect to $\prec_f$
- if $\mathtt{dec}_t\, t \neq (t', f)$, then t only has an empty decomposition

Additionally, we require that all the elementary decomposition of a given term are comparable, that is, $f[t] = f'[t'] \implies f \prec f' \vee f' \prec f \vee (f = f' \wedge t = t')$, and that $f[t] = f'[t'] \wedge f \prec f' \implies \exists v.t = v$, which effectively fixes the order of evaluation. Of course both the structure of terms and order of evaluation exist without specifying these orders: their presence, however, lets us conduct inductive reasoning without knowing the precise structure of terms, which is precisely what we need to do to prove the properties stated in Section 2.3.1. It makes this axiomatisation more interesting then the previous one, as all the proofs about semantic content of the language that one has to provide are trivial and the additional proofs about structure are also not very involved (indeed, on paper they are trivial too). In our example language, as is often the case, the properties stated earlier are self-evident for the order defined above.

The properties listed above clearly resemble these specified in [13]. They are, however, more lax in some respects: for example the order of evaluation was fixed which does not pose problems in non-formal setting, in case of such abstract axiomatisation we neither wanted to, nor even could fix the evaluation order.

As was stated before, the formalisation laid out in this section can be used to fulfil the requirements of axiomatisation of Section 2.3.2. The atomic decomposition functions and their correctness with respect to atomic plugging operation are already specified in the current formalisation, and thanks to the orders introduced above being well-founded, we can prove all the other obligations using well-founded induction with respect to both orders. The proofs are technically involved, but the idea behind them is that with intertwining induction on both the orders and the correctness of atomic decomposition functions we can simulate induction on the structure of the term. Curious reader is referred to the Coq development accompanying this thesis.

Chapter 3

From reduction semantics to abstract machine by refocusing

In this chapter, we present the refocusing transformation as it appears when using axiomatisation given in Chapter 2. We start with the reduction semantics allowing for automatic refocusing from Section 2.3.2. The transformation consists of a series of semantics, each more sophisticated then the preceding one, yet equivalent to it. We begin each section by description of the semantics – and, in case of the push-enter machine, additional requirements on the semantics – followed by statement of the equivalence theorem and a short sketch of the proof. We also provide a semantics for the arithmetic expressions language, defined in the previous chapter at each step of derivation, to better illustrate the changes. Reader curious about further details of the equivalence proofs is referred to the Coq development accompanying this thesis.

3.1 An evaluator

We begin the transformation by recalling the shape of decomposition relation, the simple generic reduce-plug-decompose wrapper, and the evaluation function using them. These are shown in Figure 3.1, where successful contraction of a potential redex is denoted with $\mapsto$, and the $\mathtt{dec}_t$ and $\mathtt{dec}_f$ functions are the elementary decomposition functions from Section 2.3.2. In Figure 3.2 the decomposition relation for arithmetic expressions is specified, with the auxiliary functions in-lined.

3.2 A pre-abstract machine

We now perform the first step of transformation, the one in which the actual *refocusing* happens. This amounts to noticing that in deterministic reduction semantics the following property should hold

$$\mathtt{dec}\, E[t][]\, d \iff \mathtt{dec}\, t\, E\, d,$$

and substituting left hand side of the equivalence for right hand side in definition of `iter` function. And indeed, this is a specific case of a property (decompose-plug), which we have asserted in Chapter 2. This transformation can be presented as in Figure 3.3, where "refocus" denotes continuing decomposition with the term obtained

$$
\begin{array}{rcl}
\mathtt{dec}\, t\, E\, d & \Longleftrightarrow & \left\{\begin{array}{ll} d = (r, E), & \text{if } \mathtt{dec}_t\, t = r \\ \mathtt{dec}_E\, E\, v\, d, & \text{if } \mathtt{dec}_t\, t = v \\ \mathtt{dec}\, t'\, (f :: E)\, d, & \text{if } \mathtt{dec}_t\, t = (t', f) \end{array}\right. \\
\mathtt{dec}_E\, [\,]\, v\, d & \Longleftrightarrow & d = v \\
\mathtt{dec}_E\, (f :: E)\, v\, d & \Longleftrightarrow & \left\{\begin{array}{ll} d = (r, E), & \text{if } \mathtt{dec}_f\, f\, v = r \\ \mathtt{dec}_E\, E\, v'\, d, & \text{if } \mathtt{dec}_f\, f\, v = v' \\ \mathtt{dec}\, t\, (f' :: E)\, d & \text{if } \mathtt{dec}_f\, f\, v = (t, f') \end{array}\right. \\
\mathtt{iter}\, v\, v' & \Longleftrightarrow & v = v' \\
\mathtt{iter}\, (r, E)\, v & \Longleftrightarrow & r \mapsto t \wedge \mathtt{dec}\, E\, [t]\, [\,]\, d \wedge \mathtt{iter}\, d\, v, \quad \text{for some } d \\
\mathtt{eval}\, t\, v & \Longleftrightarrow & \mathtt{dec}\, t\, [\,]\, d \wedge \mathtt{iter}\, d\, v, \quad \text{for some } d
\end{array}
$$

Figure 3.1: A generic evaluator for reduction semantics

$$
\begin{array}{rcl}
\mathtt{dec}\, n\, E\, d & \Longleftrightarrow & \mathtt{dec}_E\, E\, n\, d \\
\mathtt{dec}\, (t \oplus t')\, E\, d & \Longleftrightarrow & \mathtt{dec}\, t\, ([\,] \oplus t' :: E)\, d \\
 & & \\
\mathtt{dec}_E\, [\,]\, n\, d & \Longleftrightarrow & d = n \\
\mathtt{dec}_E\, ([\,] \oplus t :: E)\, n\, d & \Longleftrightarrow & \mathtt{dec}\, t\, (n \oplus [\,] :: E)\, d \\
\mathtt{dec}_E\, (m \oplus [\,] :: E)\, n\, d & \Longleftrightarrow & d = (m \oplus n, E)
\end{array}
$$

Figure 3.2: Decomposition relation for arithmetic expressions

from contraction and current context. Thus, we arrive at a more efficient evaluator, also dubbed a "pre-abstract machine" in literature [13, 6]. The definition of the machine is presented in Figure 3.4.

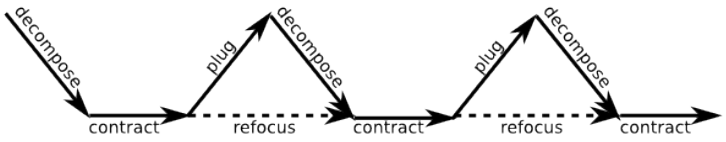

Figure 3.3: Schematic view of the refocusing transformation.

$$\begin{aligned}
\texttt{iter}_{\texttt{pam}}\, v\, v' &\iff v = v' \\
\texttt{iter}_{\texttt{pam}}\, (r, E)\, v &\iff r \mapsto t \wedge \texttt{dec}\, t\, E\, d \wedge \texttt{iter}_{\texttt{pam}}\, d\, v \quad \text{for some } d \\
\\
\texttt{eval}_{\texttt{pam}}\, t\, v &\iff \texttt{dec}\, t\, [\,]\, d \wedge \texttt{iter}_{\texttt{pam}}\, d\, v \quad \text{for some } d
\end{aligned}$$

Figure 3.4: A generic pre-abstract machine for reduction semantics

3.2.1 Correctness of derivation

The correctness of this step of transformation is captured by the following proposition.

Proposition 3.1. *For any term t and value v of a language satisfying axioms of Section 2.3.2, the equivalence* $\texttt{eval}\, t\, v \iff \texttt{eval}_{\texttt{pam}}\, t\, v$ *holds.*

The proof follows immediately from similar equivalence defined for `iter` and $\texttt{iter}_{\texttt{pam}}$, which in turn is done by induction on derivation and using the (decompose-plug) property.

3.3 A staged abstract machine

The next step of the transformation consists of, in a sense, "tying a knot" – the `dec` and $\texttt{dec}_E$ relations definitions are now made recursively dependent on the `iter` relation. This moves the semantics closer towards an abstract machine: we could even obtain an abstract machine semantics from the semantics proposed in this section but it would not resemble any abstract machine actually used due to its use of decompositions. The definition of the semantics is given in Figure 3.5. This step of derivation for arithmetic expressions is shown in Figure 3.6.

3.3.1 Correctness of derivation

The correctness of this step of derivation is again captured by a equivalence proposition:

$$
\begin{array}{rcl}
\mathtt{dec}_{\mathtt{sam}}\, t\, E\, v & \Longleftrightarrow & \begin{cases} \mathtt{iter}_{\mathtt{sam}}(r,E)\, v, & \text{if } \mathtt{dec}_t\, t = r \\ \mathtt{dec}_{E-\mathtt{sam}}\, E\, v'\, v, & \text{if } \mathtt{dec}_t\, t = v' \\ \mathtt{dec}_{\mathtt{sam}}\, t'(f :: E)\, v, & \text{if } \mathtt{dec}_t\, t = (t', f) \end{cases} \\
\mathtt{dec}_{E-\mathtt{sam}}[\,]\, v\, v' & \Longleftrightarrow & \mathtt{iter}_{\mathtt{sam}}\, v\, v' \\
\mathtt{dec}_{E-\mathtt{sam}}(f :: E)\, v\, v' & \Longleftrightarrow & \begin{cases} \mathtt{iter}_{\mathtt{sam}}(r,E)\, v', & \text{if } \mathtt{dec}_f\, f\, v = r \\ \mathtt{dec}_{E-\mathtt{sam}}\, E\, v''\, v', & \text{if } \mathtt{dec}_f\, f\, v = v'' \\ \mathtt{dec}_{\mathtt{sam}}\, t\, (f' :: E)\, v', & \text{if } \mathtt{dec}_f\, f\, v = (t, f') \end{cases} \\
\mathtt{iter}_{\mathtt{sam}}\, v\, v' & \Longleftrightarrow & v = v' \\
\mathtt{iter}_{\mathtt{sam}}(r,E)\, v & \Longleftrightarrow & r \mapsto t \wedge \mathtt{dec}_{\mathtt{sam}}\, t\, E\, v \\
\mathtt{eval}_{\mathtt{sam}}\, t\, v & \Longleftrightarrow & \mathtt{dec}\, t\, [\,]\, v
\end{array}
$$

Figure 3.5: A generic staged abstract machine for reduction semantics

$$
\begin{array}{rcl}
\mathtt{dec}\, n\, E\, v & \Longleftrightarrow & \mathtt{dec}_E\, E\, n\, v \\
\mathtt{dec}\,(t \oplus t')\, E\, v & \Longleftrightarrow & \mathtt{dec}\, t\, ([\,] \oplus t' :: E)\, v \\
\mathtt{dec}_E\, [\,]\, n\, v & \Longleftrightarrow & \mathtt{iter}\, n\, v \\
\mathtt{dec}_E\,([\,] \oplus t :: E)\, n\, v & \Longleftrightarrow & \mathtt{dec}\, t\, (n \oplus [\,] :: E)\, v \\
\mathtt{dec}_E\,(m \oplus [\,] :: E)\, n\, v & \Longleftrightarrow & \mathtt{iter}(m \oplus n, E)\, v \\
\mathtt{iter}\, n\, v & \Longleftrightarrow & n = v \\
\mathtt{iter}(m \oplus n, E)\, v & \Longleftrightarrow & \mathtt{dec}(m + n)\, E\, v
\end{array}
$$

Figure 3.6: A staged abstract machine for arithmetic expressions

Proposition 3.2. *For any term t and value v of a language satisfying axioms of Section 2.3.2, the equivalence* $\texttt{eval}_{\texttt{pam}}\, t\, v \iff \texttt{eval}_{\texttt{sam}}\, t\, v$ *holds.*

The proof is a little more complicated then in the case of the pre-abstract machine – to prove the "only if" case we need the following lemma:

Lemma 3.3. *For any term t, context E, decomposition d and value v of a language satisfying axioms of Section 2.3.2, the following hold:*

$$\begin{aligned} \texttt{dec}\, t\, E\, d \wedge \texttt{iter}_{\texttt{sam}}\, d\, v &\implies \texttt{dec}_{\texttt{sam}}\, t\, E\, v \\ \texttt{iter}_{\texttt{pam}}\, d\, v &\implies \texttt{iter}_{\texttt{sam}}\, d\, v \end{aligned}$$

The lemma is easily proved by induction on derivation of respectively $\texttt{dec}\, t\, E\, d$ and $\texttt{iter}_{\texttt{pam}}\, d\, v$. The "only if" case follows trivially. For the "if" case, we need a similar lemma

Lemma 3.4. *For any term t, context E, decomposition d and value v of a language satisfying the axioms of Section 2.3.2, the following holds:*

$$\texttt{dec}_{\texttt{sam}}\, t\, E\, v \wedge \texttt{dec}\, t\, E\, d \implies \texttt{iter}_{\texttt{pam}}\, d\, v$$

which is easily proved by induction on derivation of $\texttt{dec}_{\texttt{sam}}$, and a property stating that dec is a total function, which is also easy to prove. This suffices to prove the "if" case.

3.4 An eval-apply machine

The final part of the transformation that is applicable to any reduction semantics that allows for automatic refocusing consists of in-lining the definition of the $\texttt{iter}_{\texttt{sam}}$ relation in definitions of $\texttt{dec}_{\texttt{sam}}$ and $\texttt{dec}_{E-\texttt{sam}}$. The resulting semantics corresponds to a transition system that can be identified with an eval-apply abstract machine, operating on terms of the language. We will first describe the semantics in the form of logical relations along with the correctness result, and then extract an abstract machine semantics from the relation, again accompanied by the correctness result.

3.4.1 A relation-based semantics

The relations are constructed by in-lining of $\texttt{iter}_{\texttt{sam}}$ in $\texttt{dec}_{\texttt{sam}}$ and $\texttt{dec}_{E-\texttt{sam}}$. The resulting definition is presented in Figure 3.7. The semantics for arithmetic expressions at this step of derivation is shown in Figure 3.8.

The correctness of this step of derivation is again stated by a similar proposition:

Proposition 3.5. *For any term t and value v of a language satisfying axioms of Section 2.3.2, the equivalence* $\texttt{eval}_{\texttt{sam}}\, t\, v \iff \texttt{eval}_{\texttt{eam}}\, t\, v$ *holds.*

The proof follows by simple induction on derivations of $\texttt{dec}_{\texttt{sam}}$ and $\texttt{dec}_{\texttt{eam}}$.

3.4.2 Abstract machine semantics

As stated in Chapter 1, a deterministic abstract machine is defined as a transition function between *configurations*. The evaluation function can be defined as a transitive closure of transition function, which starts in *initial* configuration, and results in

$$
\begin{array}{rcl}
\mathtt{dec}_{\mathtt{eam}}\, t\, E\, v & \Longleftrightarrow & \left\{\begin{array}{ll} r \mapsto t' \wedge \mathtt{dec}_{\mathtt{eam}}\, t'\, E\, v, & \text{if } \mathtt{dec}_t\, t = r \\ \mathtt{dec}_{E-\mathtt{eam}}\, E\, v'\, v, & \text{if } \mathtt{dec}_t\, t = v' \\ \mathtt{dec}_{\mathtt{eam}}\, t'\, (f :: E)\, v, & \text{if } \mathtt{dec}_t\, t = (t', f) \end{array}\right. \\
\mathtt{dec}_{E-\mathtt{eam}}\, [\,]\, v\, v' & \Longleftrightarrow & v = v' \\
\mathtt{dec}_{E-\mathtt{eam}}\, (f :: E)\, v\, v' & \Longleftrightarrow & \left\{\begin{array}{ll} r \mapsto t' \wedge \mathtt{dec}_{\mathtt{eam}}\, t'\, E\, v', & \text{if } \mathtt{dec}_f\, f\, v = r \\ \mathtt{dec}_{E-\mathtt{eam}}\, E\, v''\, v', & \text{if } \mathtt{dec}_f\, f\, v = v'' \\ \mathtt{dec}_{\mathtt{eam}}\, t\, (f' :: E)\, v', & \text{if } \mathtt{dec}_f\, f\, v = (t, f') \end{array}\right. \\
\mathtt{eval}_{\mathtt{eam}}\, t\, v & \Longleftrightarrow & \mathtt{dec}_{\mathtt{eam}}\, t\, [\,]\, v
\end{array}
$$

Figure 3.7: A generic eval-apply abstract machine for reduction semantics

$$
\begin{array}{rcl}
\mathtt{dec}\, n\, E\, v & \Longleftrightarrow & \mathtt{dec}_E\, E\, n\, v \\
\mathtt{dec}\, (t \oplus t')\, E\, v & \Longleftrightarrow & \mathtt{dec}\, t\, ([\,] \oplus t' :: E)\, v \\
\mathtt{dec}_E\, [\,]\, n\, v & \Longleftrightarrow & v = n \\
\mathtt{dec}_E\, ([\,] \oplus t :: E)\, n\, v & \Longleftrightarrow & \mathtt{dec}\, t\, (n \oplus [\,] :: E)\, v \\
\mathtt{dec}_E\, (m \oplus [\,] :: E)\, n\, v & \Longleftrightarrow & \mathtt{dec}\, (m + n)\, E\, v
\end{array}
$$

Figure 3.8: An eval-apply abstract machine semantics for arithmetic expressions

a *final* configuration. Since we have two mutually recursive relations, we will introduce two configurations: an *eval* configuration, corresponding to the relation $\mathtt{dec}_{\mathtt{eam}}$, and an *apply* configuration, corresponding to the relation $\mathtt{dec}_{E-\mathtt{eam}}$. The grammar for configurations is then defined as follows:

$$c \quad ::= \quad \langle t \rangle_i \mid \langle t, E \rangle_e \mid \langle E, v \rangle_a \mid \langle v \rangle_f$$

It is now easy to specify the transition and evaluation functions. They are presented in Figure 3.9.

$$\begin{array}{rcl}
\langle t \rangle_i & \triangleright & \langle t, [\,] \rangle_e \\
\langle t, E \rangle_e & \triangleright & \begin{cases} \langle t', E \rangle_e & \text{if } \mathtt{dec}_t\, t = r \text{ and } r \mapsto t \\ \langle E, v \rangle_a & \text{if } \mathtt{dec}_t\, t = v \\ \langle t', f :: E \rangle_e & \text{if } \mathtt{dec}_t\, t = (t', f) \end{cases} \\
\langle [\,], v \rangle_a & \triangleright & \langle v \rangle_f \\
\langle f :: E, v \rangle_a & \triangleright & \begin{cases} \langle t, E \rangle_e & \text{if } \mathtt{dec}_f\, f\, v = r \text{ and } r \mapsto t \\ \langle E, v' \rangle_a & \text{if } \mathtt{dec}_f\, f\, v = v' \\ \langle t, f' :: E \rangle_e & \text{if } \mathtt{dec}_f\, f\, v = (t, f') \end{cases} \\
\mathtt{eval}_{\mathtt{peam}}\, t\, v & \Longleftrightarrow & \langle t \rangle_i \triangleright^+ \langle v \rangle_f
\end{array}$$

Figure 3.9: A generic eval-apply abstract machine semantics derived from reduction semantics

The correctness of this final step of proof is again given by a similar proposition:

Proposition 3.6. *For any term t and value v of a language satisfying axioms of Section 2.3.2, the equivalence* $\mathtt{eval}_{\mathtt{eam}}\, t\, v \iff \mathtt{eval}_{\mathtt{peam}}\, t\, v$ *holds.*

This follows, again, by simple induction, leading us to a theorem summarising the whole transformation:

Theorem 3.7. *For any term t and value v of a language satisfying axioms of Section 2.3.2, the equivalence* $\mathtt{eval}\, t\, v \iff \mathtt{eval}_{\mathtt{peam}}\, t\, v$ *holds.*

This theorem is a direct consequence of the steps shown above.

After following the last step of the derivation for the arithmetic expressions example, we obtain the machine presented in Figure 3.10.

The same machine can be obtained by applying the automatic derivation to the specification of arithmetic expressions shown in Chapter 2. However, due to the transformation being general, we have to apply some cosmetic modifications: in the obtained machine there can exist transitions that can never actually happen, or transitions that for clarity should be split in two. After in-lining the auxiliary functions and performing these minor changes we arrive at the same semantics presented above. As one can see, the obtained machine is a standard stack-based eval-apply abstract machine evaluating arithmetic expressions.

$$
\begin{array}{rcl}
\langle t \rangle_i & \rhd & \langle t, [\,] \rangle_e \\
\langle n, E \rangle_e & \rhd & \langle E, n \rangle_a \\
\langle t \oplus t', E \rangle_e & \rhd & \langle t, [\,] \oplus t' :: E \rangle_e \\
\langle [\,], n \rangle_a & \rhd & \langle n \rangle_f \\
\langle [\,] \oplus t :: E, n \rangle_a & \rhd & \langle t, n \oplus [\,] :: E \rangle_e \\
\langle n \oplus [\,] :: E, n' \rangle_a & \rhd & \langle n + n', E \rangle_e
\end{array}
$$

Figure 3.10: An eval-apply abstract machine for arithmetic expressions

3.5 A push-enter machine

The last part of transformation – which is not possible for every reduction semantics – is transition to a semantics resembling a push-enter machine and, finally, semantics given by a push-enter abstract machine. Again, we first give the semantics in logical relation form, and then extract an abstract machine from it.

3.5.1 A relation-based formulation

The transformation into a push-enter machine consists of in-lining the $\mathtt{dec}_{E-\mathtt{eam}}$ relation into $\mathtt{dec}_{\mathtt{eam}}$. This is possible only if $\mathtt{dec}_f$ never returns a value, and this is the additional property we require of the reduction semantics in order for it to be transformed into a push-enter abstract machine semantics. The resulting definition is presented in Figure 3.11.

$$
\mathtt{dec}_{\mathtt{pem}}\, t\, E\, v \iff \begin{cases}
r \mapsto t' \wedge \mathtt{dec}_{\mathtt{pem}}\, t'\, E\, v, & \text{if } \mathtt{dec}_t\, t = r \\
v = v' \wedge E = [\,] & \text{if } \mathtt{dec}_t\, t = v' \\
r \mapsto t' \wedge \mathtt{dec}_{\mathtt{pem}}\, t'\, E'\, v, & \text{if } \mathtt{dec}_t\, t = v', E = f :: E' \\
 & \text{and } \mathtt{dec}_f\, f\, v' = r \\
\mathtt{dec}_{\mathtt{pem}}\, t'\, f' :: E'\, v, & \text{if } \mathtt{dec}_t\, t = v', E = f :: E' \\
 & \text{and } \mathtt{dec}_f\, f\, v' = (t', f') \\
\mathtt{dec}_{\mathtt{pem}}\, t'\, (f :: E)\, v, & \text{if } \mathtt{dec}_t\, t = (t', f)
\end{cases}
$$

$$
\mathtt{eval}_{\mathtt{pem}}\, t\, v \iff \mathtt{dec}_{\mathtt{pem}}\, t\, [\,]\, v
$$

Figure 3.11: A generic push-enter abstract machine for reduction semantics

The correctness of this step of derivation is again stated by a similar proposition:

Proposition 3.8. *Let L be a language satisfying axioms of Section 2.3.2 and such that for any frame f and value v $\mathtt{dec}_f\, f\, v$ is not a value. For any term t and value v of such a language, the equivalence $\mathtt{eval}_{\mathtt{eam}}\, t\, v \iff \mathtt{eval}_{\mathtt{pem}}\, t\, v$ holds.*

The proof is a straightforward induction on the derivations.

We note that the arithmetic expressions used as an example throughout this chapter fulfill the additional property required to obtain a push-enter machine. The semantics resulting from following this step is shown in Figure 3.12.

$$\begin{aligned}
\mathtt{dec}\, n\, [\,]\, v &\iff v = n \\
\mathtt{dec}\, n\, ([\,] \oplus t :: E)\, v &\iff \mathtt{dec}\, t\, (n \oplus [\,] :: E)\, v \\
\mathtt{dec}\, n\, (m \oplus [\,] :: E)\, v &\iff \mathtt{dec}\, (m+n)\, E\, v \\
\mathtt{dec}\, (t \oplus t')\, E\, v &\iff \mathtt{dec}\, t\, ([\,] \oplus t' :: E)\, v
\end{aligned}$$

Figure 3.12: A push-enter abstract machine semantics for arithmetic expressions

3.5.2 Abstract machine semantics

Similarly to what we did in case of eval-apply machine, we will now state the grammar of configurations, this time only with an *eval* configuration:

$$c \quad ::= \quad \langle t \rangle_i \mid \langle t, E \rangle_e \mid \langle v \rangle_f$$

and a transition function that defines abstract machine semantics, as presented in Figure 3.13.

$$\begin{aligned}
\langle t \rangle_i \quad &\triangleright \quad \langle t, [\,] \rangle_e \\
\langle t, E \rangle_e \quad &\triangleright \quad \begin{cases}
\langle v \rangle_f & \text{if; } \mathtt{dec}_t\, t = v \text{ and } E = [\,] \\
\langle t', E \rangle_e & \text{if } \mathtt{dec}_t\, t = r \text{ and } r \mapsto t' \\
\langle t', E' \rangle_e & \text{if } \mathtt{dec}_t\, t = v, E = f :: E', \mathtt{dec}_f\, f\, v = r \text{ and } r \mapsto t \\
\langle t', f' :: E' \rangle_e & \text{if } \mathtt{dec}_t\, t = v, E = f :: E' \text{ and } \mathtt{dec}_f\, f\, v = (t', f') \\
\langle t', f :: E \rangle_e & \text{if } \mathtt{dec}_t\, t = (t', f)
\end{cases} \\
\mathtt{eval}_{\mathtt{ppem}}\, t\, v \quad &\iff \quad \langle t \rangle_i \triangleright^+ \langle v \rangle_f
\end{aligned}$$

Figure 3.13: A generic push-enter abstract machine semantics derived from reduction semantics

The correctness of this step follows analogously to the final step in the derivation of eval-apply machine, yielding the following correctness theorem:

Theorem 3.9. *Let L be a language satisfying axioms of Section 2.3.2 and such that for any frame f and value v $\mathtt{dec}_f\, f\, v$ is not a value. For any term t and value v of L, the equivalence $\mathtt{eval}\, t\, v \iff \mathtt{eval}_{\mathtt{ppem}}\, t\, v$ holds.*

The proof follows straightforwardly from the series of lemmas shown in previous steps.

The extraction of abstract machine from the semantics of arithmetic expressions specified in previous section yields the machine presented in Figure 3.14.

$$
\begin{array}{rcl}
\langle t \rangle_i & \rhd & \langle t, [\,] \rangle_e \\
\langle n, [\,] \rangle_e & \rhd & \langle n \rangle_f \\
\langle n, [\,] \oplus t :: E \rangle_e & \rhd & \langle t, n \oplus [\,] :: E \rangle_e \\
\langle n, n' \oplus [\,] :: E \rangle_e & \rhd & \langle n + n', E \rangle_e \\
\langle t \oplus t', E \rangle_e & \rhd & \langle t, [\,] \oplus t' :: E \rangle_e
\end{array}
$$

Figure 3.14: A push-enter abstract machine for arithmetic expressions

As was the case with the eval-apply machine, the semantics given above is identical to the one resulting from in-lining the auxiliary functions and performing some cosmetic modifications in the machine obtained by the automatic transformation. It is also a standard push-enter version of stack-based machine evaluating arithmetic expressions.

Chapter 4

Case studies

In this chapter we present three case studies of using the transformation as presented in Chapter 3. We begin with deriving both eval-apply and push-enter machines for standard lambda calculus with call-by-value reduction strategy in Section 4.1, then proceed to derive both types of machines for lambda calculus with call-by-name strategy in Section 4.2. Transformations of these lambda calculi have been shown in [13], should the reader like to follow them step by step: here we shall only show how the languages fulfil the axioms detailed in Chapter 2, and show the resulting machines. Finally in Section 4.3 we provide a more sophisticated example by presenting the derivation of an abstract machine for a simple MiniML programming language.

4.1 Lambda calculus under call-by-value

As a first example we present the lamda calulus with call-by-value reduction strategy and left-to-right evaluation order. The grammars of terms, values, potential redexes and context frames are given as:

$$
\begin{array}{ll}
t ::= x \mid \lambda x.t \mid t\ t & \text{(terms)} \\
v ::= x \mid \lambda x.t & \text{(values)} \\
r ::= v\ v & \text{(redexes)} \\
f ::= [\,]\,t \mid v\,[\,] & \text{(context frames)}
\end{array}
$$

The operation of plugging a term into a context frame is the obvious function replacing the hole in the frame with a term. The requirements on decompositions of values and redexes are met, following a similar argument as in the example in Chapter 2. We define contraction as a standard capture avoiding substitution:

$$(\lambda x.t)\,v \quad \mapsto \quad t[x/v]$$

Now we can define the atomic decomposition functions. They are specified below:

$$\begin{aligned} \mathtt{dec}_t\, v &= v \\ \mathtt{dec}_t(t_1\, t_2) &= (t_1, [\,]t_2) \end{aligned}$$

$$\begin{aligned} \mathtt{dec}_f([\,]t)\, v &= (t, v[\,]) \\ \mathtt{dec}_f(v[\,])\, v' &= (v\, v') \end{aligned}$$

The correctness of these definitions is self-evident. We also note, that the additional property from Section 3.5 is met, so we will be able to derive a push-enter semantics for this calculus. The only thing left to be done is specifying the order on context frames. The definition becomes evident if we look at the requirements it has to conform to. We define it as the smallest strict order relation $\prec$, such that for any term t and value v, $v[\,] \prec [\,]t$. It is easy to show that this definition satisfies the required properties.

In this way we have fixed the input semantics in a form that allows us to plug it into the automatic transformation. Below, we show both eval-apply and push-enter machines that are derived from this semantics. Of course – as stated before in Section 3.4 – due to generality of derivation, in the obtained machine there can exist empty transitions or a transition that should be split in two for clarity: these cosmetic modifications, along with in-lining of $\mathtt{dec}_t$, $\mathtt{dec}_f$ and contraction functions, are all that needs to be done after the automatic part of the transformation.

The eval-apply machine presented obtained by refocusing is shown below. As mentioned in [13], the machine corresponds to the well known CK machine by Felleisen and Friedman[17].

$$\begin{aligned} \langle t \rangle_i &\rhd \langle t, [\,] \rangle_e \\ \langle t_1\, t_2, E \rangle_e &\rhd \langle t_1, [\,]t_2 :: E \rangle_e \\ \langle v, E \rangle_e &\rhd \langle E, v \rangle_a \\ \langle [\,], v \rangle_a &\rhd \langle v \rangle_f \\ \langle [\,]t :: E, v \rangle_a &\rhd \langle t, v[\,] :: E \rangle_e \\ \langle (\lambda x.t)[\,] :: E, v \rangle_a &\rhd \langle t[v/x], E \rangle_e \end{aligned}$$

Below we also present a push-enter variant of an abstract machine for call-by-value lambda calculus. To the best of our knowledge, this machine does not correspond to any of the well known abstract machines for this calculus.

$$\begin{aligned} \langle t \rangle_i &\rhd \langle t, [\,] \rangle_e \\ \langle t_1\, t_2, E \rangle_e &\rhd \langle t_1, [\,]t_2 :: E \rangle_e \\ \langle v, [\,] \rangle_e &\rhd \langle v \rangle_f \\ \langle v, [\,]t :: E \rangle_e &\rhd \langle t, v[\,] :: E \rangle_e \\ \langle v, (\lambda x.t)[\,] :: E \rangle_e &\rhd \langle t[v/x], E \rangle_e \end{aligned}$$

This case study demonstrates the use of the transformation for a simple language – albeit, of course, much more complicated then the example accompanying the trans-

formation. After another relatively straightforward example we will proceed to a more involved study.

4.2 Lambda calculus under call-by-name

The second example presented here is a call-by-name lambda calculus. The terms and values are identical to the ones in the previous example: the difference lies in the definition of context frames and redexes. All the grammars are presented below:

$$
\begin{array}{rr}
t ::= x \mid \lambda x.t \mid t\ t & \text{(terms)} \\
v ::= x \mid \lambda x.t & \text{(values)} \\
r ::= v\ t & \text{(redexes)} \\
f ::= [\,]t & \text{(context frames)}
\end{array}
$$

The contraction is, again, defined as a capture avoiding substitution:

$$(\lambda x.t_1)t_2 \quad \mapsto \quad t_1[t_2/x]$$

The atomic decomposition functions are, again, quite similar to the case of call-by-value, with difference lying in the decomposition of frames:

$$
\begin{array}{rcl}
\mathtt{dec}_t\, x & = & x \\
\mathtt{dec}_t(\lambda x.t) & = & \lambda x.t \\
\mathtt{dec}_t(t_1\, t_2) & = & (t_1, [\,]t_2)
\end{array}
$$

$$\mathtt{dec}_f([\,]t)\, v \quad = \quad (v\ t)$$

The correctness of these definitions is, again, self-evident. We define the relation $\prec$ on context frames as an empty relation, and note that it trivially fulfils the obligations on this relation. We also note, that the language above satisfies the additional obligation required in order to derive a push-enter machine from the language.

This way we have fixed the language – we now proceed to obtain eval-apply and push-enter abstract machine semantics for it. As in the previous section, these machines undergo some cosmetic changes and in-lining of the auxiliary functions. We first present the eval-apply abstract machine obtained:

$$
\begin{array}{rcl}
\langle t \rangle_i & \rhd & \langle t, [\,] \rangle_e \\
\langle t_1\, t_2, E \rangle_e & \rhd & \langle t_1, [\,]t_2 :: E \rangle_e \\
\langle v, E \rangle_e & \rhd & \langle E, v \rangle_a \\
\langle [\,], v \rangle_a & \rhd & \langle v \rangle_f \\
\langle [\,]t :: E, \lambda x.t' \rangle_a & \rhd & \langle t'[t/x], E \rangle_e
\end{array}
$$

Below, we present the push-enter abstract machine obtained from the transformation – as always, with some cosmetic modifications – which, as was noted in [13] corresponds to a variant of Krivine's abstract machine using substitutions.

$$\begin{array}{rcl}
\langle t \rangle_i & \rhd & \langle t, [\,] \rangle_e \\
\langle t_1\, t_2, E \rangle_e & \rhd & \langle t_1, [\,] t_2 :: E \rangle_e \\
\langle v, [\,] \rangle_e & \rhd & \langle v \rangle_f \\
\langle \lambda x.t_1, [\,] t_2 :: E \rangle_e & \rhd & \langle t_1[t_2/x], E \rangle_e
\end{array}$$

After presenting another example of automatic derivation on a rather simple language, we proceed to the more involved example.

4.3 MiniML

The MiniML, being somewhat of a middle ground between the lambda calculus and fully fledged functional languages, is a common example on which one can present some more sophisticated features of programming languages. Based on the call-by-value lambda calculus the language provides, in the version presented here, also let-expressions, least fixed point operator and natural numbers constructors with a pattern matching operator **case**. The constructors are eager, evaluating their arguments, and order of evaluation is left-to-right. The grammars of terms, values, potential redexes and context frames are presented below:

$$\begin{array}{ll}
t ::= x \mid \lambda x.t \mid t\ t \mid \mathtt{0} \mid S t \mid \mathbf{let}\, x = t\, \mathbf{in}\, t \mid \mathbf{fix}\, x = t \mid \mathbf{case}\, t\, \mathbf{of}\, t\,(x, t) & \text{(terms)} \\
v ::= x \mid \lambda x.t \mid S v \mid \mathtt{0} & \text{(values)} \\
r ::= v\, v \mid \mathbf{let}\, x = v\, \mathbf{in}\, t \mid \mathbf{fix}\, x = t \mid \mathbf{case}\, v\, \mathbf{of}\, t\,(x, t) & \text{(redexes)} \\
f ::= [\,] t \mid v [\,] \mid S [\,] \mid \mathbf{let}\, x = [\,]\, \mathbf{in}\, t \mid \mathbf{case}\, [\,]\, \mathbf{of}\, t\,(x, t) & \text{(frames)}
\end{array}$$

Plugging is defined, as in the previous examples, by replacing the hole in the frame with the term. The requirements on decompositions of values and redexes are also met, although we now have to use an inductive argument to prove it, due to the S constructor. The `case` expression takes three parameters: the expression that should be evaluated and the alternatives for `0` and S. The second alternative binds the variable to the argument of S. Formally, the contraction is defined as:

$$\begin{array}{rcl}
(\lambda x.t)\, v & \mapsto & t[v/x] \\
\mathbf{let}\, x = v\, \mathbf{in}\, t & \mapsto & t[v/x] \\
\mathbf{fix}\, x = t & \mapsto & t[\mathtt{fix}\, x = t/x] \\
\mathbf{case}\, \mathtt{0}\, \mathbf{of}\, t_1\,(x, t_2) & \mapsto & t_1 \\
\mathbf{case}\, S v\, \mathbf{of}\, t_1\,(x, t_2) & \mapsto & t_2[v/x]
\end{array}$$

The atomic decomposition functions are, as the rest of the language, extensions of call-by-value lambda calculus to the new constructions. They are defined as:

$$
\begin{array}{rcl}
\mathtt{dec}_t\, x & = & x \\
\mathtt{dec}_t\, (\lambda x.t) & = & \lambda x.t \\
\mathtt{dec}_t\, \mathtt{0} & = & \mathtt{0} \\
\mathtt{dec}_t\, (S t) & = & (t, S[\,]) \\
\mathtt{dec}_t\, (\mathbf{let}\, x = t_1\, \mathbf{in}\, t_2) & = & (t_1, \mathbf{let}\, x = [\,]\, \mathbf{in}\, t_2) \\
\mathtt{dec}_t\, (\mathbf{case}\, t_1\, \mathbf{of}\, t_2\, (x, t_3)) & = & (t_1, \mathbf{case}\, [\,]\, \mathbf{of}\, t_2\, (x, t_3)) \\
\mathtt{dec}_t\, (\mathbf{fix}\, x = t) & = & \mathbf{fix}\, x = t \\
\mathtt{dec}_t\, (t_1\, t_2) & = & (t_1, [\,] t_2) \\
 & & \\
\mathtt{dec}_f\, ([\,] t)\, v & = & (t, v[\,]) \\
\mathtt{dec}_f\, (v[\,])\, v' & = & (v\, v') \\
\mathtt{dec}_f\, (S[\,])\, v & = & S v \\
\mathtt{dec}_f\, (\mathbf{let}\, x = [\,]\, \mathbf{in}\, t)\, v & = & \mathbf{let}\, x = v\, \mathbf{in}\, t \\
\mathtt{dec}_f\, (\mathbf{case}\, [\,]\, \mathbf{of}\, t_1\, (x, t_2))\, v & = & \mathbf{case}\, v\, \mathbf{of}\, t_1\, (x, t_2)
\end{array}
$$

The correctness of these definitions is self-evident. Note that due to the rule for the frame $S[\,]$, the definition of $\mathtt{dec}_f$ does not fulfil the requirements of Section 3.5, so we will only be able to obtain an eval-apply machine. The only thing left to be done is specifying the order $\prec$ on context frames. However, we may notice that only the application can be split into two different pairs of term and context frame, so the ordering very similar to the one for call-by-value lambda calculus will be appropriate. Actually, we define it in the exact same manner as in Section 4.1: as the smallest strict order relation $\prec$, such that for any term t and value v, $v[\,] \prec [\,]t$. It is easy to show that this definition satisfies the required properties.

This way, we have fixed the semantics in a form which we can plug into the transformation. After in-lining the auxiliary functions and cosmetic modifications analogous to the ones needed in the other case studies, we arrive at the abstract machine presented in Figure 4.1.

As is evident from this case study, the mechanisation of the transformation lets us easily obtain abstract machines for more sophisticated languages, where transformation by hand could be error prone and tedious. Also, note that this example justifies the need for some possible outcomes for atomic decomposition functions: the possibility that a term is a redex that cannot be decomposed – which is the case for the `fix` operator – and that a value in a context is also a value – which shows up with the S constructor or, in truth, in any eager datatype constructor. The implementation of this case study in Coq sports also a pair constructor and projection functions, however, we left them out of this presentation to avoid unnecessary clutter.

$$
\begin{array}{rcl}
\langle t \rangle_i & \rhd & \langle t, [\,] \rangle_e \\
\langle t_1\, t_2, E \rangle_e & \rhd & \langle t_1, [\,]\, t_2 :: E \rangle_e \\
\langle x, E \rangle_e & \rhd & \langle E, x \rangle_a \\
\langle \lambda x.t, E \rangle_e & \rhd & \langle E, \lambda x.t \rangle_a \\
\langle \mathbf{let}\, x = t_1\, \mathbf{in}\, t_2, E \rangle_e & \rhd & \langle t_1, \mathbf{let}\, x = [\,]\, \mathbf{in}\, t_2 :: E \rangle_e \\
\langle \mathbf{case}\, t_1\, \mathbf{of}\, t_2\, (x, t_3), E \rangle_e & \rhd & \langle t_1, \mathbf{case}\, [\,]\, \mathbf{of}\, t_2\, (x, t_3) :: E \rangle_e \\
\langle \mathbf{fix}\, x = t, E \rangle_e & \rhd & \langle t\, [\mathbf{fix}\, x = t/x], E \rangle_e \\
\langle \mathsf{0}, E \rangle_e & \rhd & \langle E, \mathsf{0} \rangle_a \\
\langle S\, t, E \rangle_e & \rhd & \langle t, S\, [\,] :: E \rangle_e \\
\langle [\,], v \rangle_a & \rhd & \langle v \rangle_f \\
\langle [\,]\, t :: E, v \rangle_a & \rhd & \langle t, v\, [\,] :: E \rangle_e \\
\langle (\lambda x.t)\, [\,] :: E, v \rangle_a & \rhd & \langle t\, [v/x], E \rangle_e \\
\langle S\, [\,] :: E, v \rangle_a & \rhd & \langle E, S\, v \rangle_a \\
\langle \mathbf{let}\, x = [\,]\, \mathbf{in}\, t :: E, v \rangle_a & \rhd & \langle t\, [v/x], E \rangle_e \\
\langle \mathbf{case}\, [\,]\, \mathbf{of}\, t_1\, (x, t_2) :: E, \mathsf{0} \rangle_a & \rhd & \langle t_1, E \rangle_e \\
\langle \mathbf{case}\, [\,]\, \mathbf{of}\, t_1\, (x, t_2) :: E, S\, v \rangle_a & \rhd & \langle t_2 [v/x], E \rangle_e
\end{array}
$$

Figure 4.1: An eval-apply abstract machine for MiniML obtained by the refocusing transformation

Chapter 5

Implementation in the Coq proof assistant

The transformation, as defined in Chapter 3, along with case studies of Chapter 4 and the extensions studied in detail in Chapters 6-8 is implemented in the Coq proof assistant and proved correct. Thanks to the formalisation being engineered as a framework, one can use it to transform their own reduction semantics to possibly interesting abstract machines. This is achieved by using the system of modules provided by Coq[7], which is an extension of the well known module system of the ML family of languages, such as OCaml or SML.

The first part of the Coq implementation of the transformation are definitions of the module types (or signatures in SML parlance) of the reduction semantics. The module types as implemented in Coq allow not only for declarations of datatypes or functions – as is the case in its weaker-typed cousins – but also of arbitrary properties required of these values. Thus, we can define a series of module types mirroring the axiomatisation laid out in Chapter 2: first a type of the source language, then – building on it – the type of generic reduction semantics, and finally – by refining some aspects of the reduction semantics – the type of semantics ready for refocusing. We also extend the last of these types with additional property required to obtain the push-enter machine, specify the type of a generic abstract machine and provide – again, building on the source language definition – the type of strict semantics allowing for refocusing.

The last challenge of this part of implementation is to prove that one can always obtain a reduction semantics ready for refocusing, should one starting with a module satisfying the type of strict semantics. This is done using functors – constructs that, when given a module of a specific type, produce a module of another type. The functors in Coq are extended versions of their namesakes found in SML and OCaml. We implement such a functor, which generally follows the description provided in Chapter 2, and in this way finish the first part of implementation: we are now ready to implement the refocusing transformation.

The next – and biggest in terms of the sheer size – part of the development is the implementation of the actual transformation. Each step of the transformation is given a separate module type, describing the shape of decomposition and evaluation functions. Throughout the development, these functions are represented as inductively defined propositions, which is a standard technique of implementation of such

functions in Coq. Then for each step of the transformation, we implement a functor that implements it. The functor takes the reduction semantics allowing for automatic application of refocusing – that is, a module realising the type of such semantics – as an input. It then uses a functor that implements an earlier stage of derivation to reach the step that should now be implemented. After this preparations, the functor proceeds to implement the type of the next step of derivation – and a proof that this step has not changed the semantics. Completing this proof allows us to assert that the semantics obtained in the resulting module is equivalent to that of reduction semantics given as an input. These proofs follow the outline given in Chapter 3, though the proofs are generally split into smaller lemmas, as this makes work with the proof assistant easier. As a result of this step of development, we get functors that are able to build an abstract machine from reduction semantics. In particular, the functor that generates the push-enter machine requires the input reduction semantics to provide a proof that the decomposition of a context frame never leads to a value, as described in Section 3.5.

The final part of the development is an implementation of a wide range of case studies. Only by considering a wide enough selection of languages we can be assured that the axiomatisation is sensible. Each of the case studies consists of two parts. First, we need to implement the modules of types required to enable the automatic refocusing. This proved not to be a difficult part of the development, though more then once specific languages – most notably the most complicated of them, the MiniML – made us reconsider the axioms and loosen some obligations. This has led to an axiomatisation that is, in our opinion, not overly complicated while providing the possibility of refocusing for a wide range of languages. The other part of each of the case studies was to transform the machine obtained by the automatic procedure into a more explicit form (by in-lining the $\texttt{dec}_t$, $\texttt{dec}_f$ and contraction functions, and eliminating some transitions that could never be reached), and proving the equivalence of these machines – which was an easy task, as such changes do not alter the computation conducted by the machine. In addition, most case studies provide an additional definition of reduction semantics for the language under consideration, which is the standard reduction semantics for the calculus. These semantics are accompanied with proofs that they are equivalent to the semantics used in automatic refocusing – as the semantics used in refocusing are given in a slightly different form – and this way we can assert correctness of the machine obtained by the transformation with respect to the well-known reduction semantics.

Below, we present two fragments of implementation. In Figure 5.1, we show the transition function of a generic eval-apply abstract machine, along with its transitive closure and the evaluation function of the machine. In Figure 5.2, we present the relation-based version of generic eval-apply machine. In both of those figures, `dec_term` and `dec_context` are the functions $\texttt{dec}_t$ and $\texttt{dec}_f$ used throughout the formalisation.

```
Inductive transition :
  configuration → configuration → Prop :=
| t_init : forall t,
             ⟨t⟩_i ▷ ⟨t, []⟩_e
| t_red  : forall t t0 E r,
             (DT    : dec_term t = R.in_red r)
             (CONTR : R.contract r = Some t0),
             ⟨t, E⟩_e ▷ ⟨t0, E⟩_e
| t_val  : forall t v E,
             (DT    : dec_term t = R.in_val v),
             ⟨t, E⟩_e ▷ ⟨E, v⟩_a
| t_term : forall t t0 f E
             (DT    : dec_term t = R.in_term t0 f),
             ⟨t, E⟩_e ▷ ⟨t0, f :: E⟩_e
| t_cfin : forall v,
             ⟨[], v⟩_a ▷ ⟨v⟩_f
| t_cred : forall v t f E r,
             (DC    : dec_context f v = R.in_red r)
             (CONTR : R.contract r = Some t),
             ⟨f :: E, v⟩_a ▷ ⟨t, E⟩_e
| t_cval : forall v v0 f E,
             (DC    : dec_context f v = R.in_val v0),
             ⟨f :: E, v⟩_a ▷ ⟨E, v0⟩_a
| t_cterm: forall v t f f0 E,
             (DC    : dec_context f v= R.in_term t f0),
             ⟨f :: E, v⟩_a ▷ ⟨t, f0 :: c⟩_e
where " a ▷ b " := (transition a b).

Inductive trans_close :
  configuration → configuration → Prop :=
| one_step   : forall c0 c1,
             c0 ▷ c1 →c0 ▷^+ c1
| multi_step : forall c0 c1 c2,
             c0 ▷ c1 →
             c1 ▷^+ c2 →
             c0 ▷^+ c2
where " a ▷^+ b " := (trans_close a b).

Inductive eval : term → value → Prop :=
| e_intro : forall t v,
             ⟨t⟩_i ▷^+ ⟨v⟩_f →
             eval t v.
```

Figure 5.1: A Coq definition of an eval-apply abstract machine, along with its evaluation function.

```
Inductive dec : R.term → R.context → R.value → Prop :=
| d_d_r  : forall t t0 E r v
             (DT    : dec_term t = R.in_red r)
             (CONTR : R.contract r = Some t0)
             (R_T   : dec t0 E v),
             dec t E v
| d_v    : forall t E v v0
             (DT    : dec_term t = R.in_val v)
             (R_C   : decctx v E v0),
             dec t E v0
| d_term : forall t t0 f E v
             (DT    : dec_term t = R.in_term t0 f)
             (R_T   : dec t0 (f :: E) v),
             dec t c v
with decctx : R.value → R.context → R.value → Prop :=
| dc_end : forall v,
             decctx v [] v
| dc_red : forall v v0 f E r t
             (DC    : dec_context f v = R.in_red r)
             (CONTR : R.contract r = Some t)
             (R_T   : dec t E v0),
             decctx v (f :: E) v0
| dc_rec : forall v v0 v1 f E
             (DC    : dec_context f v = R.in_val v0)
             (R_C   : decctx v0 E v1),
             decctx v (f :: E) v1
| dc_trm : forall v v0 f f0 E t
             (DC    : dec_context f v = R.in_term t f0)
             (R_T   : dec t (f0 :: E) v0),
             decctx v (f :: E) v0.

Inductive eval : R.term → R.value → Prop :=
| e_intro : forall t v,
             dec t [] v →
             eval t v.
```

Figure 5.2: A relational definition of generic eval-apply machine in Coq

Chapter 6

Refocusing in environment-based reduction semantics

An extension of the refocusing transformation in a way allowing for extraction of abstract machines with environments is due to Biernacka and Danvy[6]. In their paper, they have minimally extended the Curien's calculus of closures $\lambda\rho$[10] in such a way, that one-step reduction is definable in it. Then, after following standard refocusing until arriving at an abstract machine, they are able to shortcut some transitions in order to get back to standard $\lambda\rho$ calculus and, finally, unfold the definition of closures to arrive at an environment-based machine.

The calculi of closures – or, as they were introduced by Curien, of *explicit substitutions* – were specifically designed to try and bridge the gap between the abstract machines with environments and the lambda calculi. In most applications, the abstract machine should be as efficient as possible, and performing an actual substitution every time a redex is contracted is both inefficient and hard to compile down to a virtual machine – which causes the environment machines to be very popular. On the other hand, lambda calculus is the most standard way of describing the high-level programming languages. The calculi of closures considerably narrow the gap between the two, and are used in many applications[10, 20].

We extend our formalisation to allow for refocusing of languages utilising explicit substitutions by first modifying the axiomatisation to suit closure calculi in Section 6.1 and then describing the additional two steps of the transformation that are applicable in this formalisation. In Section 6.2 we describe how to use the additional axioms to shortcut some intermediate transitions and how to unfold the closures and we remark on additional properties required to arrive at a push-enter machine. We finally show two case studies, arriving at two standard environment-based machines in Section 6.3.

6.1 Calculi of closures: changes to axiomatisation

The extension of refocusing to environment-based machines requires a considerable amount of changes, so it is worth noting that the axiomatisation of Chapter 2 could

also be used with closure calculi. However, the additional steps of automatic transformation presented in this chapter would not be possible, and the user wishing to obtain an environment-based machine would need to do these transformation by hand. On the other hand, the axiomatisation presented here uses some of the most sophisticated features of the type system of Coq, which makes it harder to use. Below we present the changes to the axiomatisation along with the example of a $\lambda\rho$ calculus under call-by-value.

6.1.1 Basic changes

As in Chapter 2, we need to specify the syntactic categories used in the transformation. We will use two closure calculi, C and $\widehat{C}$, both over the same set T of terms, with the closures of C being the subset of closures of $\widehat{C}$. The calculus $\widehat{C}$ will fulfil the axioms of Chapter 2 – and it will be transformed analogously to the transformation of Chapter 3 into an abstract machine. The smaller calculus C will need to be in such a form, that the closures can be unfolded in order to arrive at an environment-based machine. The closures of the calculi are denoted respectively with c and $\hat{c}$, with $c \subseteq \hat{c}$. We also define values in each calculus, with $v \subseteq c$, $\hat{v} \subseteq \hat{c}$ and $v \subseteq \hat{v}$, as well as context frames f, and $\hat{f}$, again with $f \subseteq \hat{f}$. As it has to be possible to automatically apply refocusing to the calculus $\widehat{C}$, we also specify potential redexes in it, denoted with $\hat{r}$. The requirements on values, redexes and decompositions specified in Chapter 2 also required here. As an example, we present a $\lambda\rho$ calculus under call-by-value. The grammars are defined as follows:

$$t ::= i \mid \lambda t \mid t\ t \qquad \text{(terms)}$$

$$\begin{aligned}
c &::= t[s] \quad \text{where } s ::= \text{list } c && \text{(closures of } C\text{)}\\
v &::= (\lambda t)[s] && \text{(values of } C\text{)}\\
f &::= v[\,] \mid [\,]c && \text{(frames of } C\text{)}
\end{aligned}$$

$$\begin{aligned}
\hat{c} &::= \hat{c}\ \hat{c} \mid t[\hat{s}] \quad \text{where } \hat{s} ::= \text{list } \hat{c} && \text{(closures of } \widehat{C}\text{)}\\
\hat{v} &::= (\lambda t)[\hat{s}] && \text{(values of } \widehat{C}\text{)}\\
\hat{f} &::= \hat{v}[\,] \mid [\,]\hat{c} && \text{(frames of } \widehat{C}\text{)}\\
\hat{r} &::= i[\hat{s}] \mid (t\ t)[\hat{s}] \mid \hat{v}\ \hat{v} && \text{(redexes of } \widehat{C}\text{)}
\end{aligned}$$

with the contraction function defined as

$$\begin{aligned}
i[\hat{c}_0 \cdot \hat{c}_1 \cdots \hat{c}_i \cdots \hat{c}_n] &\mapsto \hat{c}_i\\
(t_1\ t_2)[\hat{s}] &\mapsto (t_1[\hat{s}])(t_2[\hat{s}])\\
(\lambda t[\hat{s}])\,\hat{v} &\mapsto t[\hat{v} \cdot \hat{s}]
\end{aligned}$$

We use deBruijn indices in this formalisation, so a λ-abstraction simply binds the occurrence of index `0` in its body. The closures, values and context frames of the C

calculus correspond to the $\lambda\rho$ calculus, while those of $\widehat{C}$ – to its minimal extension, $\lambda\hat{\rho}$ as defined in [6]. We can see that a function plug – working over the larger calculus – can be naturally defined, and that this calculus fulfils all the requirements laid out both in this section and in Chapter 2.

Both the calculi in our example use *explicit substitutions* – which are denoted with s and $\hat{s}$ respectively – a construction that allows to avoid using a meta-level substitution function. As a matter of fact, one more requirement we make in the axiomatisation is that the calculi use explicit substitutions – that is, that a term with such a substitution is a closure. Furthermore, we require that closures of the target calculus, C, are either terms with substitutions or values. This property lets us unfold these closures at the final step of the transformation. The last requirement is that the closures of target calculus only have empty decompositions. Our calculi satisfy also these requirements.

6.1.2 Compatibility and changes to further parts of axiomatisation

For the most part, the rest of the axiomatisation is analogous to the one of Chapter 2, with terms replaced by closures of $\widehat{C}$. However, the requirements of previous section are not strong enough to capture the required connection between the calculi. Below, we specify the additional properties that solve this problem. These requirements are implemented in Coq as strongly specified, dependently typed functions that are to be provided by the user.

$$\begin{aligned}
\mathtt{dec}_t\, c = \hat{r} \wedge \hat{r} \mapsto \hat{c}' &\implies \exists c'.c' = \hat{c}' \vee \exists c', f.f[c'] = \hat{c}' \\
\mathtt{dec}_t\, c = \hat{v} &\implies \exists v.v = \hat{v} \\
\mathtt{dec}_f\, f\, v = \hat{r} \wedge \hat{r} \mapsto \hat{c} &\implies \exists c.c = \hat{c} \vee \exists c, f.f[c] = \hat{c}' \\
\mathtt{dec}_f\, f\, v = \hat{v}' &\implies \exists v'.v' = \hat{v}' \\
\mathtt{dec}_f\, f\, v = (\hat{c}', \hat{f}') &\implies \exists c', f'.f' = \hat{f}' \wedge c' = \hat{c}'
\end{aligned}$$

The one case not covered above – if $\mathtt{dec}_t\, c = (\hat{c}', \hat{f}')$ can never appear, as it would require c to have a non-empty decomposition, which we have already assumed to be impossible. We should now show how these requirements are satisfied in our example calculus, in order to provide some intuition: we will, however, only show how the first implication holds, as the other are analogous.

We begin by observing when can a closure c of $\lambda\rho$ be considered a redex of $\lambda\hat{\rho}$ – it can happen only if $c = i[s]$, or $c = t_1\, t_2[s]$. In the first case, if the contraction is successful, it would choose the i-th closure of the substitution s, let it be $\hat{c}_i$. However, all the closures in s are actually closures of $\lambda\rho$, with the i-th one being c_i. Of course $c_i = \hat{c}_i$, so the requirement holds. The latter case is more interesting: note that in this case the contraction always succeeds, producing a closure $\hat{c}' = (t_1[s])(t_2[s])$. We now observe that this new closure will be decomposed in one step to a closure $c' = t_1[s]$, which is a closure of $\lambda\rho$, and a context frame $f' = [\,]\, t_2[s]$ – also of the target calculus. Of course, the equality $f'[c'] = \hat{c}'$ holds, so the requirement is also fulfilled in this case. The proofs of the other compatibility requirements follows the same path, while the other properties required of the semantics are defined analogously to the case study concerning the standard lambda-calculus under call-by-value, presented in Section 4.1.

6.2 Additional steps toward an efficient eval-apply machine

As mentioned before, the transformation follows the same steps that are outlined in Chapter 3 up to obtaining an eval-apply machine defined using relations. This machine works on closures, however, and had we transformed it into an abstract machine semantics, we would have obtained a machine using closures instead of environments. However, our extended axiomatisation allows us to make two further steps. For the reference, below we include the eval-apply machine working on closures of $\widehat{C}$, as obtained via the standard refocusing transformation, and then proceed through the two additional changes that finally let us arrive at an abstract machine using environments.

$$\begin{array}{rcl}
\mathtt{dec}_{\mathtt{eam}}\,\hat{c}\,\hat{E}\,v & \Longleftrightarrow & \begin{cases} \hat{r} \mapsto \hat{c}' \wedge \mathtt{dec}_{\mathtt{eam}}\,\hat{c}'\,\hat{E}\,v, & \text{if } \mathtt{dec}_t\,\hat{c} = \hat{r} \\ \mathtt{dec}_{E-\mathtt{eam}}\,\hat{E}\,\hat{v}'\,v, & \text{if } \mathtt{dec}_t\,\hat{c} = \hat{v}' \\ \mathtt{dec}_{\mathtt{eam}}\,\hat{c}'\,(\hat{f} :: \hat{E})\,\hat{v}, & \text{if } \mathtt{dec}_t\,\hat{c} = (\hat{c}', \hat{f}) \end{cases} \\
\mathtt{dec}_{E-\mathtt{eam}}\,[\,]\,\hat{v}\,v' & \Longleftrightarrow & \hat{v} = v' \\
\mathtt{dec}_{E-\mathtt{eam}}\,(\hat{f} :: \hat{E})\,\hat{v}\,v' & \Longleftrightarrow & \begin{cases} \hat{r} \mapsto \hat{c}' \wedge \mathtt{dec}_{\mathtt{eam}}\,\hat{c}'\,\hat{E}\,v', & \text{if } \mathtt{dec}_f\,\hat{f}\,\hat{v} = \hat{r} \\ \mathtt{dec}_{E-\mathtt{eam}}\,\hat{E}\,\hat{v}''\,v', & \text{if } \mathtt{dec}_f\,\hat{f}\,\hat{v} = \hat{v}'' \\ \mathtt{dec}_{\mathtt{eam}}\,\hat{c}\,(\hat{f}' :: \hat{E})\,v', & \text{if } \mathtt{dec}_f\,\hat{f}\,\hat{v} = (\hat{c}, \hat{f}') \end{cases} \\
\mathtt{eval}_{\mathtt{eam}}\,t\,v & \Longleftrightarrow & \mathtt{dec}_{\mathtt{eam}}\,t \bullet [\,]\,v
\end{array}$$

6.2.1 A move to the target calculus C

We are now ready to exploit the specific connection between the two calculi used in the axiomatisation and proceed to the machine using closures of the target calculus. In order to do this, we use the properties asserted in the previous section – and we arrive at the following definition:

$$\begin{array}{rcl}
\mathtt{dec}_{\mathtt{eamc}}\,c\,E\,v & \Longleftrightarrow & \begin{cases} \mathtt{dec}_{\mathtt{eamc}}\,c'\,(E' \cdot E)\,v, & \text{if } \mathtt{dec}_t\,c = \hat{r}, \hat{r} \mapsto \hat{c}' \\ & \quad \text{and } E'[c] = \hat{c}' \\ \mathtt{dec}_{E-\mathtt{eamc}}\,E\,v'\,v, & \text{if } \mathtt{dec}_t\,c = \hat{v}' \text{ and } v' = \hat{v}' \end{cases} \\
\mathtt{dec}_{E-\mathtt{eamc}}\,[\,]\,v\,v' & \Longleftrightarrow & v = v' \\
\mathtt{dec}_{E-\mathtt{eamc}}\,(f :: E)\,v\,v' & \Longleftrightarrow & \begin{cases} \mathtt{dec}_{\mathtt{eamc}}\,c'\,(E' \cdot E)\,v', & \text{if } \mathtt{dec}_f\,f\,v = \hat{r}, \hat{r} \mapsto \hat{c}' \\ & \quad \text{and } E'[c] = \hat{c}' \\ \mathtt{dec}_{E-\mathtt{eamc}}\,E\,v''\,v', & \text{if } \mathtt{dec}_f\,f\,v = \hat{v}'' \\ & \quad \text{and } v'' = \hat{v}'' \\ \mathtt{dec}_{\mathtt{eamc}}\,c\,(f' :: E)\,v', & \text{if } \mathtt{dec}_f\,f\,v = (\hat{c}, \hat{f}'), \\ & \quad c = \hat{c} \text{ and } f' = \hat{f}' \end{cases} \\
\mathtt{eval}_{\mathtt{eamc}}\,t\,v & \Longleftrightarrow & \mathtt{dec}_{\mathtt{eamc}}\,(t[\bullet])\,[\,]\,v
\end{array}$$

Note that in this definition the context E' appearing after reductions can be either empty or have exactly one frame, as specified in the previous section. The existence of E', as well as the other objects of the calculus C that appear in this step of the transformation, was asserted in the compatibility requirements stated in the axiomatisation. The correctness of this step of the derivation is summarised by the following proposition:

Proposition 6.1. *For any term t and value v of a language satisfying axioms of Section 6.2, the equivalence $\mathtt{eval}_{\mathtt{eam}}\, t\, v \iff \mathtt{eval}_{\mathtt{eamc}}\, t\, v$ holds.*

The proof follows by routine induction, using the additional properties required in Section 6.2.

6.2.2 Unfolding the closures

The last step in transition to an eval-apply machine using environments is unfolding the closures. To do this, we exploit the fact that each closure of the target calculus is either a term with a substitution, or a value. Thus, we may check which one of those cases has occurred when we reach a closure, and proceed accordingly: either checking the structure of the term if we have one, or proceeding with checking the context if the closure was a value. This process can be summarised by following definition:

$$\mathtt{dec}_{\mathtt{ueam}}\, t\, s\, E\, v \iff \begin{cases} \mathtt{dec}_{\mathtt{ueam}}\, t'\, s'\, E' \cdot E\, v, & \text{if } \mathtt{dec}_t\, c = \hat{r}, \hat{r} \mapsto \hat{c}' \\ & \text{and } E'[t'[s']] = \hat{c}' \\ \mathtt{dec}_{\mathtt{ueam}}\, E' \cdot E\, v'\, v, & \text{if } \mathtt{dec}_t\, c = \hat{r}, \hat{r} \mapsto \hat{c}' \\ & \text{and } E'[v'] = \hat{c}' \\ \mathtt{dec}_{E-\mathtt{ueam}}\, E\, v'\, v, & \text{if } \mathtt{dec}_t c = \hat{v}' \text{ and } v' = \hat{v}' \end{cases}$$

$$\mathtt{dec}_{E-\mathtt{ueam}}\, [\,]\, v\, v' \iff v = v'$$

$$\mathtt{dec}_{E-\mathtt{ueam}}\, (f :: E)\, v\, v' \iff \begin{cases} \mathtt{dec}_{\mathtt{ueam}}\, t\, s\, E' \cdot E\, v', & \text{if } \mathtt{dec}_f\, f\, v = \hat{r}, \hat{r} \mapsto \hat{c}' \\ & \text{and } E'[t[s]] = \hat{c}' \\ \mathtt{dec}_{\mathtt{ueam}}\, E' \cdot E\, v''\, v', & \text{if } \mathtt{dec}_f\, f\, v = \hat{r}, \hat{r} \mapsto \hat{c}' \\ & \text{and } E'[v''] = \hat{c}' \\ \mathtt{dec}_{E-\mathtt{ueam}}\, E\, v''\, v', & \text{if } \mathtt{dec}_f\, f\, v = \hat{v}'' \\ & \text{and } v'' = \hat{v}'' \\ \mathtt{dec}_{\mathtt{ueam}}\, t\, s\, (f' :: E)\, v', & \text{if } \mathtt{dec}_f\, f\, v = (\hat{c}, \hat{f}'), \\ & t[s] = \hat{c} \text{ and } f' = \hat{f}' \\ \mathtt{dec}_{\mathtt{ueam}}\, (f' :: E)\, v''\, v', & \text{if } \mathtt{dec}_f\, f\, v = (\hat{c}, \hat{f}'), \\ & v'' = \hat{c} \text{ and } f' = \hat{f}' \end{cases}$$

$$\mathtt{eval}_{\mathtt{ueam}}\, t\, v \iff \mathtt{dec}_{\mathtt{ueam}}\, t \bullet [\,]\, v$$

This semantics builds on the previous one by simply checking which of the two possible forms a closure has, and proceeding to the right alternative. The correctness of this step of transformation is given in the following proposition.

Proposition 6.2. *For any term t and value v of a language satisfying axioms of Section 6.2, the equivalence $\mathtt{eval}_{\mathtt{eamc}}\, t\, v \iff \mathtt{eval}_{\mathtt{ueam}}\, t\, v$ holds.*

The proof is yet again done by induction on derivations of $\mathtt{dec}_{\mathtt{eamc}}$ and $\mathtt{dec}_{\mathtt{ueam}}$. What is important, now the terms of the language and the substitutions appear explicitly in the relation $\mathtt{dec}_{\mathtt{ueam}}$, allowing us to extract from it a machine using the substitution as an environment, which was our goal in this formalisation. This is done analogously to the last step in the basic refocusing transformation, as presented in Chapter 3. We arrive at the following definition of the machine's configurations:

$$c \quad ::= \quad \langle t \rangle_i \mid \langle t, s, E \rangle_e \mid \langle E, v \rangle_a \mid \langle v \rangle_f$$

and of the transitions and evaluation function:

$$\begin{array}{rcl}
\langle t \rangle_i & \rhd & \langle t, \bullet, [\,] \rangle_e \\
\langle t, s, E \rangle_e & \rhd & \begin{cases} \langle t', s', E' \cdot E \rangle_e & \text{if } \mathtt{dec}_t\, t[s] = \hat{r}, \hat{r} \mapsto \hat{c} \text{ and } E'[t'[s']] = \hat{c} \\ \langle E' \cdot E, v \rangle_a & \text{if } \mathtt{dec}_t\, t[s] = \hat{r}, \hat{r} \mapsto \hat{c} \text{ and } E'[v] = \hat{c} \\ \langle E, v \rangle_a & \text{if } \mathtt{dec}_t\, t[s] = v \end{cases} \\
\langle [\,], v \rangle_a & \rhd & \langle v \rangle_f \\
\langle f :: E, v \rangle_a & \rhd & \begin{cases} \langle t, s, E \rangle_e & \text{if } \mathtt{dec}_f\, f\, v = \hat{r}, \hat{r} \mapsto \hat{c} \text{ and } E'[t[s]] = \hat{c} \\ \langle E' \cdot E, v' \rangle_a & \text{if } \mathtt{dec}_f\, f\, v = \hat{r}, \hat{r} \mapsto \hat{c} \text{ and } E'[v'] = \hat{c} \\ \langle E, v' \rangle_a & \text{if } \mathtt{dec}_f\, f\, v = v' \\ \langle t, s, f' :: E \rangle_e & \text{if } \mathtt{dec}_f\, f\, v = (t[s], f') \\ \langle f' :: E, v' \rangle_a & \text{if } \mathtt{dec}_f\, f\, v = (v', f') \end{cases} \\
\mathtt{eval}_{\mathtt{peam}}\, t\, v & \iff & \langle t \rangle_i \rhd^+ \langle v \rangle_f
\end{array}$$

As before, the context E' can be at most one frame long due to the constraints of Section 6.1. In some places, where we believed it would not cause confusion, the transition to the target calculus was in-lined more then in the previous semantics – though this is only a measure of notational convenience. The equivalence of this representation to the former one is proved in exactly the same way as in Chapter 3, yielding a summarising theorem:

Theorem 6.3. *For any term t and value v of a language satisfying axioms of Section 2.3.2, the equivalence* $\mathtt{eval}\, t\, v \iff \mathtt{eval}_{\mathtt{peam}}\, t\, v$ *holds.*

A remark on values in the target calculus

The reader might notice that in the example that has accompanied the changes to the axiomatisation all the values were of the form $t[s]$ and wonder why should we allow other values – especially as this makes the resulting machine much bigger. This is indeed somewhat of a nuisance, as values of other structure are really rare. However, they do appear at times – a notable example arises if we equip the language with a control operator that captures the current evaluation context. In this case, such a captured context is definitely a value, yet it cannot be seen as a term with a substitution. It could also occur with the eager constructors though this example was not studied in detail.

6.2.3 Toward a push-enter machine

As we recall from Chapter 3, there was an additional requirement that the semantics had to fulfil if we wanted to derive a push-enter machine. This final step of transformation consisted of in-lining of the $\mathtt{dec}_{E-\mathtt{eam}}$ relation in the definition of $\mathtt{dec}_{\mathtt{eam}}$ relation. In order to be able to do this, we needed to be sure that at any given moment of evaluation, we will look at most at the innermost frame of the reduction context. This requirement was formalised in terms of the $\mathtt{dec}_f$ function: it could never return a value, as that would spark another inspection of the reduction context. In case of the calculi of closure this requirement still stands. However, it is not the only requirement we have to make: because the closure may be an arbitrary value, not only of the form of a term with a substitution, a chain of context inspection may build up. This is why, in addition to the requirement carried over from the basic variant of the transformation, we require that the closures of the target language only be of the form $t[s]$. Now, we can start with the semantics defined in the previous section and follow the steps outlined in Chapter 3, to arrive at the following definition:

$$\mathtt{dec}_{\mathtt{upem}}\, t\, s\, E\, v \iff \begin{cases} \mathtt{dec}_{\mathtt{upem}}\, t'\, s'\, E' \cdot E\, v, & \text{if } \mathtt{dec}_t\, t[s] = \hat{r}, \hat{r} \mapsto \hat{c}' \\ & \text{and } E'[t'[s']] = \hat{c}' \\ v = v' \wedge E = [\,] & \text{if } \mathtt{dec}_t\, t[s] = v' \\ \mathtt{dec}_{\mathtt{upem}}\, t'\, s'\, E'' \cdot E'\, v, & \text{if } \mathtt{dec}_t\, t[s] = v', \hat{r} \mapsto \hat{c}', \\ & E = f :: E', \mathtt{dec}_f\, f\, v' = \hat{r} \\ & \text{and } E''[t'[s']] = \hat{c}' \\ \mathtt{dec}_{\mathtt{pem}}\, t'\, s'\, f' :: E'\, v, & \text{if } \mathtt{dec}_t\, t[s] = v', E = f :: E' \\ & \text{and } \mathtt{dec}_f\, f\, v' = (t'[s'], f') \end{cases}$$

$$\mathtt{eval}_{\mathtt{upem}}\, t\, v \iff \mathtt{dec}_{\mathtt{upem}}\, t \bullet [\,]\, v$$

We summarise the correctness of this step of derivation with the following proposition:

Proposition 6.4. *Let L be a language satisfying axioms of Section 2.3.2 and such that for any frame $\hat{f}$ and value $\hat{v}$ $\mathtt{dec}_f\, \hat{f}\, \hat{v}$ is not a value and that for any closure c, it is of the form $t[s]$ where t is a term and s – a substitution. For any term t and value v of L, the equivalence $\mathtt{eval}_{\mathtt{ueam}}\, t\, v \iff \mathtt{eval}_{\mathtt{upem}}\, t\, v$ holds.*

Once again the proof is a straightforward induction on derivations of $\mathtt{dec}_{\mathtt{ueam}}$ and $\mathtt{dec}_{\mathtt{upem}}$. This final step lets us finally derive a push-enter machine with environment. The grammar of configuration and the definition of the machine follow.

$$c \quad ::= \quad \langle t \rangle_i \mid \langle t, s, E \rangle_e \mid \langle v \rangle_f$$

$$
\begin{array}{lcl}
\langle t \rangle_i & \triangleright & \langle t, \bullet, [\,] \rangle_e \\
\langle t, s, E \rangle_e & \triangleright & \begin{cases}
\langle v \rangle_f & \text{if } \mathtt{dec}_t\, t[s] = v \text{ and } E = [\,] \\
\langle t', s', E' \cdot E \rangle_e & \text{if } \mathtt{dec}_t\, t[s] = \hat{r},\ \hat{r} \mapsto \hat{c} \text{ and } E'[t'[s']] = \hat{c} \\
\langle t', s', E'' \cdot E \rangle_e & \text{if } \mathtt{dec}_t\, t[s] = v,\ \mathtt{dec}_f\, f\, v = \hat{r},\ \hat{r} \mapsto \hat{c} \text{ and } E''[t'[s']] = \hat{c} \\
\langle t', s', f' :: E' \rangle_e & \text{if } \mathtt{dec}_t\, t[s] = v \text{ and } \mathtt{dec}_f\, f\, v = (t'[s'], f')
\end{cases}
\end{array}
$$

$$
\mathtt{eval}_{\mathtt{ppem}}\, t\, v \iff \langle t \rangle_i \rhd^+ \langle v \rangle_f
$$

The correctness of this step follows analogously to the final step in derivation of eval-apply machine, yielding the following correctness theorem:

Theorem 6.5. *Let L be a language satisfying axioms of Section 2.3.2 and such that for any frame $\hat{f}$ and value $\hat{v}$ $\mathtt{dec}_f\, \hat{f}\, \hat{v}$ is not a value and that for any closure c, it is of the form $t[s]$ where t is a term and s – a substitution. For any term t and value v of L, the equivalence $\mathtt{eval}\, t\, v \iff \mathtt{eval}_{\mathtt{ppem}}\, t\, v$ holds.*

The proof follows straightforwardly from the series of lemmas shown in previous steps.

6.3 Case studies

We finish the presentation of this extension of refocusing transformation with two case studies. First, we apply the transformation to the $\lambda\rho$ calculus with call-by-value evaluation strategy, deriving an eval-apply machine coinciding with the CEK machine. Then we take to the $\lambda\rho$ calculus with call-by-name order of evaluation and derive a push-enter machine, which coincides with the Krivine's machine.

6.3.1 The $\lambda\rho$ calculus with call-by-value

As this calculus was already discussed as an example while presenting the axiomatisation, we will just provide the definitions here as a reference:

$$t ::= i \mid \lambda t \mid t\ t \qquad \text{(terms)}$$

$$\begin{aligned}
c &::= t[s] \quad \text{where } s ::= \text{list } c && \text{(closures of } C\text{)}\\
v &::= \lambda t[s] && \text{(values of } C\text{)}\\
f &::= v[\,] \mid [\,]c && \text{(frames of } C\text{)}
\end{aligned}$$

$$\begin{aligned}
\hat{c} &::= \hat{c}\,\hat{c} \mid t[\hat{s}] \quad \text{where } \hat{s} ::= \text{list } \hat{c} && \text{(closures of } \widehat{C}\text{)}\\
\hat{v} &::= \lambda t[\hat{s}] && \text{(values of } \widehat{C}\text{)}\\
\hat{f} &::= \hat{v}[\,] \mid [\,]\hat{c} && \text{(frames of } \widehat{C}\text{)}\\
\hat{r} &::= i[\hat{s}] \mid t\ t[\hat{s}] \mid \hat{v}\,\hat{v} && \text{(redexes of } \widehat{C}\text{)}
\end{aligned}$$

Contracting the redexes amounts to taking the appropriate closure from the substitution, changing application into composition of closures or extending the substitution, depending on the redex:

$$\begin{aligned}
i[\hat{c}_0 \cdot \hat{c}_1 \cdots \hat{c}_i \cdots \hat{c}_n] &\mapsto \hat{c}_i\\
t_1\,t_2[\hat{s}] &\mapsto t_1[\hat{s}]\,t_2[\hat{s}]\\
(\lambda t[\hat{s}])\,\hat{v} &\mapsto t[\hat{v} \cdot \hat{s}]
\end{aligned}$$

The atomic decomposition functions are

$$\begin{aligned}
\mathtt{dec}_t\,(\hat{c}_1\,\hat{c}_2) &= (\hat{c}_1, [\,]\hat{c}_2)\\
\mathtt{dec}_t\,i[\hat{s}] &= i[\hat{s}]\\
\mathtt{dec}_t\,(\lambda t[\hat{s}]) &= \lambda t[\hat{s}]\\
\mathtt{dec}_t\,(t_1\,t_2[\hat{s}]) &= t_1\,t_2[\hat{s}]
\end{aligned}$$

$$\begin{aligned}
\mathtt{dec}_f\,([\,]\hat{c})\,\hat{v} &= (\hat{c}, \hat{v}[\,])\\
\mathtt{dec}_f\,(\hat{v}[\,])\,\hat{v}' &= \hat{v}\,\hat{v}'
\end{aligned}$$

As we have already elaborated on the correctness of these definitions in Section 6.1 and in Chapter 4, we omit the details. Just as the context frames virtually identical to the ones of lambda calculus using call-by-value reduction strategy, so is the relation $\prec$, which lets us realise all the obligations.

After fixing the language suitable for refocusing in this way, we present the eval-apply machine resulting from the transformation. As before, the atomic decompositions and contraction are in-lined in the transitions of the machine, and empty transitions are eliminated. As was already discovered in [6], the machine obtained in this way corresponds to the well-known CEK machine.

$$
\begin{array}{rcl}
\langle t \rangle_i & \rhd & \langle t, \bullet, [\,] \rangle_e \\
\langle t_1\, t_2, s, E \rangle_e & \rhd & \langle t_1, s, [\,]\, t_2[s] :: E \rangle_e \\
\langle i, s, E \rangle_e & \rhd & \langle t, s', E \rangle_e, \text{ where } i[s] \mapsto t[s'] \\
\langle \lambda t, s, E \rangle_e & \rhd & \langle E, \lambda t[s] \rangle_a \\
\langle [\,], v \rangle_a & \rhd & \langle v \rangle_f \\
\langle [\,] t[s] :: E, v \rangle_a & \rhd & \langle t, s, v[\,] :: E \rangle_e \\
\langle (\lambda t[s])[\,] :: E, v \rangle_a & \rhd & \langle t, v \cdot s, E \rangle_e
\end{array}
$$

6.3.2 The $\lambda\rho$ calculus with call-by-name

We now proceed to the second case study – we begin with another variant of the $\lambda\rho$ calculus, this time evaluated with call-by-name strategy. The differences between this calculus and our previous example are similar to those in standard lambda calculus: the changes occur only in the shape of the evaluation contexts and redexes – and, obviously, in the contraction function. The grammars take on the following form:

$$
\begin{array}{ll}
t ::= i \mid \lambda t \mid t\, t & \text{(terms)} \\
\\
c ::= t[s] \quad \text{where } s ::= \text{list } c & \text{(closures of } C\text{)} \\
v ::= \lambda t[s] & \text{(values of } C\text{)} \\
f ::= [\,] c & \text{(frames of } C\text{)} \\
\\
\hat{c} ::= \hat{c}\, \hat{c} \mid t[\hat{s}], \text{ where } \hat{s} ::= \text{list } \hat{c} & \text{(closures of } \widehat{C}\text{)} \\
\hat{v} ::= \lambda t[\hat{s}] & \text{(values of } \widehat{C}\text{)} \\
\hat{f} ::= [\,] \hat{c} & \text{(frames of } \widehat{C}\text{)} \\
\hat{r} ::= i[\hat{s}] \mid t\, t[\hat{s}] \mid \hat{v}\, \hat{c} & \text{(redexes of } \widehat{C}\text{)}
\end{array}
$$

while contraction differs from the previous example only to match the form of the β-redex:

$$
\begin{array}{rcl}
i[\hat{c}_0 \cdot \hat{c}_1 \cdots \hat{c}_i \cdots \hat{c}_n] & \mapsto & \hat{c}_i \\
t_1\, t_2[\hat{s}] & \mapsto & t_1[\hat{s}]\, t_2[\hat{s}] \\
(\lambda t[\hat{s}])\, \hat{c} & \mapsto & t[\hat{c} \cdot \hat{s}]
\end{array}
$$

The correctness of this definitions follow from the same arguments as in the previous example. The atomic decomposition functions are also defined similarly:

$$
\begin{array}{rcl}
\mathtt{dec}_t\,(\hat{c}_1\,\hat{c}_2) & = & (\hat{c}_1, [\,]\hat{c}_2) \\
\mathtt{dec}_t\,i[\hat{s}] & = & i[\hat{s}] \\
\mathtt{dec}_t\,(\lambda t[\hat{s}]) & = & \lambda t[\hat{s}] \\
\mathtt{dec}_t\,(t_1\,t_2[\hat{s}]) & = & t_1\,t_2[\hat{s}] \\
\\
\mathtt{dec}_f\,([\,]\hat{c})\,\hat{v} & = & \hat{v}\,\hat{c}
\end{array}
$$

We now can define the order $\prec$ on evaluation contexts as an empty relation, just as was the case with lambda calculus with call-by-name evaluation strategy. The correctness of these definitions, as well as the additional properties stated in Section 6.1 is easy to prove.

We are now ready to use refocusing to derive an abstract machine from the semantics above. Note that we can obtain a push-enter machine from this language, as both additional obligations are also fulfilled: the closures of target calculus are only of the shape $t[s]$, and the decomposition of context frames never yields a value. After the same cosmetic modifications as were applied in the other case studies, the derivation yields the following machine:

$$
\begin{array}{rcl}
\langle t \rangle_i & \rhd & \langle t, \bullet, [\,] \rangle_e \\
\langle t_1\,t_2, s, E \rangle_e & \rhd & \langle t_1, s, [\,]t_2[s] :: E \rangle_e \\
\langle i, s, E \rangle_e & \rhd & \langle t, s', E \rangle_e, \text{where } i[s] \mapsto t[s'] \\
\langle \lambda t, s, [\,] \rangle_e & \rhd & \langle \lambda t[s] \rangle_f \\
\langle \lambda t, s, [\,]c :: E \rangle_e & \rhd & \langle t, c \cdot s, E \rangle_e
\end{array}
$$

This machine coincides with the Krivine's machine using explicit substitutions, as presented in [10]. This result was already observed by Biernacka and Danvy in their work on application of refocusing to the calculi of closures[6].

Chapter 7

Refocusing in context-sensitive reduction semantics

The notion of context-sensitive reduction semantics were first introduced by Biernacka and Danvy in [6] in order to deal with applying refocusing transformation to languages utilising multiple binders. They also showed that this notion can be used to obtain an abstract machine for the language with control effects such as call/cc or shift and reset operators[5]. However, it has always been used in conjunction with the extension to the environment-based languages. In this section, we will enhance the formalisation of Chapter 2 to allow for the context-sensitive reduction, and demonstrate its use with two case studies: a substitution based lambda calculus with call/cc operator under call-by-value, and an environment based lambda calculus under call-by-value with right-to-left evaluation.

7.1 Context-sensitive reduction semantics and its formalisation

In standard reduction semantics contraction has type $R \to T$. This, however, can be insufficient for certain languages utilising more sophisticated constructs, where contraction depends not only on the redex, but also on the shape of the whole context. Obviously, the standard refocusing transformation cannot be applied to such languages. This can be easily fixed, though, by changing the type of contraction into $R \times E \to T \times E$. Such formulation of reduction semantics allows for refocusing, as no part of the transformation depends on any specific properties of contraction.

The changes in formalisation to allow for context-sensitive reduction are minor. In axiomatisation, they consist only of change to the type of contraction and propagating this change in `dec`, `iter` and `eval` relations. In the proper transformation, the change of type of contraction is propagated through definitions and proofs without any impact on the structure or difficulty of proofs. Beside introducing this extension in the standard transformation, a combination with the environment-based extension is provided, as this is the setting in which context-sensitive reduction semantics have appeared and is a source of many interesting case studies.

7.2 Application in languages with control operators

As a first case study of context-sensitive refocusing, we consider the substitution-based lambda calculus with call/cc operator. This operator, first introduced in the Scheme programming language, allows to pass the current continuation as an argument of the function called. The language is evaluated under call-by-value in left-to-right order. Instead of actually adding the call/cc operator, we add two operators that together easily simulate call/cc, $\mathcal{A}$ and $\mathcal{C}$ as introduced by Felleisen and Friedman[17]. The term call/cc(t) is equivalent to $\mathcal{C}(\lambda z.z(tz))$, where z is not free in t, which will become evident when we give the reduction rules for these constructs. This extension yields the following grammars for terms, values, redexes and stack frames.

$$\begin{array}{rcl} t & ::= & x \mid \lambda x.t \mid t\,t \mid \mathcal{A}\,t \mid \mathcal{C}\,t \\ v & ::= & x \mid \lambda x.t \\ r & ::= & v\,v \mid \mathcal{A}\,t \mid \mathcal{C}\,t \\ f & ::= & [\,]t \mid v[\,] \end{array}$$

As one can see, the new operators, when applied to terms, also constitute potential redexes – and to reduce them we will need to use the context. We also provide a contraction rule for the call/cc operator. The context-sensitive reduction is defined as follows:

$$\begin{array}{rcl} ((\lambda x.t)v, E) & \mapsto & (t[v/x], E) \\ (\mathcal{A}\,t, E) & \mapsto & (t, [\,]) \\ (\mathcal{C}\,t, E) & \mapsto & (t\ \lambda z.\mathcal{A}E[z], [\,]) \text{ where } z \text{ is fresh} \\ (\text{call/cc}\,t, E) & \mapsto & (t\ \lambda z.\mathcal{A}E[z], E) \text{ where } z \text{ is fresh} \end{array}$$

This way, the $\mathcal{C}$ operator captures the current evaluation context as a function, while the $\mathcal{A}$ operator simply discards current context. The atomic decomposition functions are straightforward to define:

$$\begin{array}{rcl} \mathtt{dec}_t\,v & = & v \\ \mathtt{dec}_t\,\mathcal{A}\,t & = & \mathcal{A}\,t \\ \mathtt{dec}_t\,\mathcal{C}\,t & = & \mathcal{C}\,t \\ \mathtt{dec}_t(t_1\,t_2) & = & (t_1, [\,]t_2) \\ \\ \mathtt{dec}_f([\,]t)\,v & = & (t, v[\,]) \\ \mathtt{dec}_f(v[\,])\,v' & = & (v\,v') \end{array}$$

These functions obviously fulfil the obligations needed for refocusing. As the addition of the operators $\mathcal{A}$ and $\mathcal{C}$ did not change the context frames, we can use the same ordering that was used for the lambda calculus under call-by-value.

We see that we can now plug the obtained semantics into the context-sensitive version of the refocusing transformation. The resulting machine, after some cosmetic

changes and in-lining of $\mathtt{dec}_t$ and $\mathtt{dec}_f$ functions, we obtain the following abstract machine.

$$
\begin{array}{rcl}
\langle t \rangle_i & \rhd & \langle t, [\,] \rangle_e \\
\langle t_1\, t_2, E \rangle_e & \rhd & \langle t_1, [\,]\, t_2 :: E \rangle_e \\
\langle v, E \rangle_e & \rhd & \langle E, v \rangle_a \\
\langle \mathscr{A}\, t, E \rangle_e & \rhd & \langle t, [\,] \rangle_e \\
\langle \mathscr{C}\, t, E \rangle_e & \rhd & \langle t\ \lambda z.\mathscr{A}\, E[z], [\,] \rangle_e \text{ where } z \text{ is fresh}
\end{array}
$$

$$
\begin{array}{rcl}
\langle [\,], v \rangle_a & \rhd & \langle v \rangle_f \\
\langle [\,] t :: E, v \rangle_a & \rhd & \langle t, v[\,] :: E \rangle_e \\
\langle (\lambda x.t)[\,] :: E, v \rangle_a & \rhd & \langle t[v/x], E \rangle_e
\end{array}
$$

We note that this machine does not correspond to the CK machine with $\mathscr{C}$ and $\mathscr{A}$ operators as presented in [17]. However, it still is an extension of the CK machine. The difference is caused by contracting the $\mathscr{C}$ operator to the form of a λ-term, instead of explicitly representing the continuations.

7.3 Application in languages with multiple binders

The second case study focuses on combination of the two extensions. We consider a $\lambda\rho$ calculus with multiple binders and right-to-left, call-by-value evaluation scheme. We then extend it, using the axiomatisation given in Chapter 6, to a $\lambda\hat{\rho}$ calculus in which one step reduction can be defined. This yields the following grammars for terms, values, closures and context frames in each calculus, and potential redexes:

$$
\begin{array}{rcl}
t & ::= & i \mid \lambda^n.t \mid t\ t \mid \\
c & ::= & t[s] \quad \text{where } s ::= \text{list } c \\
v & ::= & \lambda^n.t[s] \\
f & ::= & c[\,] \mid [\,]v \\
\hat{c} & ::= & \hat{c}\,\hat{c} \mid t[\hat{s}] \quad \text{where } \hat{s} ::= \text{list } \hat{c} \\
\hat{v} & ::= & \lambda^n.t[\hat{s}] \\
\hat{f} & ::= & \hat{c}[\,] \mid [\,]\hat{v} \\
\hat{r} & ::= & i[\hat{s}] \mid t\ t[\hat{s}] \mid v\ v
\end{array}
$$

This formalisation uses deBruijn indices for variables, and multiple binders per lambda abstraction: $\lambda^n t$ binds $n+1$ variables, which are the indices numbered $\mathtt{0}$ to n in the term t. As one can see, this definition fulfils the requirements stated in Section 6.1, save for the compatibility requirements. To show those, however, we first need to define the contraction function – which will be context-sensitive in order to effectively deal with multiple binders in lambda abstraction. The definition is as follows.

$$\begin{array}{rcl}
(i[\hat{c}_0 \cdot \hat{c}_1 \cdots \hat{c}_i \cdots \hat{c}_n], \hat{E}) & \mapsto & (\hat{c}_i, \hat{E}) \\
(t_1\, t_2[\hat{s}], \hat{E}) & \mapsto & (t_1[\hat{s}]\, t_2[\hat{s}], \hat{E}) \\
((\lambda^n.t[\hat{s}])\, \hat{v}_0, []\hat{v}_1 :: \cdots :: []\hat{v}_n :: \hat{E}) & \mapsto & (t[\hat{v}_n \cdots \hat{v}_1 \cdot \hat{v}_0 \cdot \hat{s}], \hat{E})
\end{array}$$

As is usual for such languages [6], we could choose whether a lambda abstraction binding n arguments, applied only to $k < n$ arguments should be stuck – which would be the case for the definition above – or should it bind the arguments it is given and decrease the number of variables bound. However crucial it might be for the designer of the machine, it is of little importance while considering refocusing, as the transformation barely depends on the contraction function.

The rest of the implementation is very similar to the one for context-free call-by-value calculus presented in Chapter 6. The atomic decomposition functions are defined as follows:

$$\begin{array}{rcl}
\mathtt{dec}_t\,(\hat{c}_1\, \hat{c}_2) & = & (\hat{c}_2, \hat{c}_1[\,]) \\
\mathtt{dec}_t\, i[\hat{s}] & = & i[\hat{s}] \\
\mathtt{dec}_t\,(\lambda^n t[\hat{s}]) & = & \lambda^n t[\hat{s}] \\
\mathtt{dec}_t\,(t_1\, t_2[\hat{s}]) & = & t_1\, t_2[\hat{s}]
\end{array}$$

$$\begin{array}{rcl}
\mathtt{dec}_f\,(\hat{c}[\,])\,\hat{v} & = & (\hat{c}, []\hat{v}) \\
\mathtt{dec}_f\,([]\hat{v'})\,\hat{v} & = & \hat{v}\,\hat{v'}
\end{array}$$

The correctness of these functions is self-evident. The only thing that is different then in the example presented in the previous chapter is the order $\prec$ on evaluation contexts, which now is defined as the smallest relation such that for any $\hat{c}$ and $\hat{v}$, $[]\hat{v} \prec \hat{c}[\,]$. The functions defining compatibility between the calculi we consider now are also analogous to the ones appearing there.

Now we are able to plug this semantics into a refocusing transformation. After some cosmetic changes and in-lining of $\mathtt{dec}_t$ and $\mathtt{dec}_f$ functions (which, just as in other case studies on environments-based machines, is rather tedious to prove due to abundance of dependent types), we arrive at the following push-enter machine.

$$\begin{array}{rcll}
\langle t \rangle_i & \rhd & \langle t, \bullet, [\,] \rangle_e & \\
\langle i, c_0 \cdot c_1 \cdots (t[s])_i \cdots c_n, E \rangle_e & \rhd & \langle t, s, E \rangle_e & \\
\langle t_1\, t_2, s, E \rangle_e & \rhd & \langle t_2, s, t_1[s][\,] :: E \rangle_e & \\
\langle \lambda^n t, s, [\,] \rangle_e & \rhd & \langle \lambda^n t[s] \rangle_f & \\
\langle \lambda^n t, s, t[s'][\,] :: E \rangle_e & \rhd & \langle t, s', [\,]\lambda^n t[s] :: E \rangle_e & \\
\langle \lambda^n t, s, []v :: E \rangle_e & \rhd & \langle t, s', E' \rangle_e & \text{if } (\lambda^n t[s]\, v, E) \mapsto (s', E')
\end{array}$$

In the above semantics we have avoided in-lining the contraction of beta-redex to avoid excessive clutter. As we may see, this machine coincides with a version of Leroy's Zinc machine in a version operating on terms (while in the original definition it used an instruction set)[25].

Chapter 8

Refocusing and nontermination

The last extension of refocusing that we cover in this thesis is refocusing for potentially nonterminating programs. The chapter is structured as follows. In Section 8.1 we briefly discuss semantics for nonterminating programs; this is followed by Section 8.2, in which we outline the changes in the formalisation and transformation. Finally in Section 8.3 we note the different implementation techniques required by this extension. As the axiomatisation does not change in this extensions, and the transition systems resulting from the transformation are also very similar to those obtained from normal refocusing – though interpreted in a different manner – we will not provide case studies. This extension is completely orthogonal to the previous two, it could be used just as well for closure calculi or languages utilising context-sensitive reduction. However, at present there is no implementation for these combinations of extensions – this is left as a part of the future work.

8.1 Reduction semantics for nonterminating programs

As the understanding of computer programs – and their abundance in daily life – continues to grow, the difference between a diverging program and a program that goes raw grows stark. In many applications the program should never terminate, and in some cases premature termination of a program could be of dire consequences. Most of the standard techniques do not let us reason about diverging programs or, at the very least, make it very inconvenient. For semantics in which evaluation relations are defined as a transitive closure of some single-step transition function – such as, for example, abstract machines or reduction semantics – this can be done easier then for others, by using coinductive reasoning and a variation of execution traces [26, 28].

8.2 Formalising coinductive refocusing

The axiomatisation of Chapter 2 is not changed at all, as it only has to do with one-step reduction. The change allowing for reasoning about nonterminating programs is made to the evaluator. The `dec` relation is specified just as in Figure 3.1, but the other relations are defined differently. We follow the approach of [26] for the most

part, both when it comes to the idea and the basics of Coq implementation – the main difference is that Leroy and Grall store the whole term in the execution trace, while we store decompositions. This gives us more information by also providing us with the exact place in the term where the contraction happens. The resulting definitions are given below, but this time they should be interpreted coinductively – as the *largest* relations satisfying the equations.

$$\begin{aligned} \texttt{gather}\, v\, s &\iff s = v \cdot \circ \\ \texttt{gather}\,(r,E)((r,E)\cdot s) &\iff r \mapsto t \wedge \texttt{dec}\, E[t]\,[\,]\, d \wedge \texttt{gather}\, d\, s, \text{ for some } d \end{aligned}$$

$$\texttt{eval}\, t\, s \iff \texttt{dec}\, t\,[\,]\, d \wedge \texttt{gather}\, d\, s, \text{ for some } d$$

Here, the execution trace s is an element of the greatest fixed point of the following grammar – a potentially infinite stream of decompositions.

$$s ::= \circ \mid d \cdot s,$$

The rest of the transformation follows strictly the steps defined in Chapter 3, with evaluation relation and, since staged abstract machine, $\texttt{dec}$ and $\texttt{dec}_E$ relations, defined coinductively. The proofs are very similar to the ones presented there, but the inductive reasoning should be replaced by coinduction. The transition to machines again changes types again, as the machine's transition beside moving to the next configuration, optionally emits a decomposition if one was reached. This allows us to replace the standard transitive closure of the transition function with function that additionally gathers the emitted decompositions into a stream. The definitions of this special transitive closure and the evaluation function are specified below:

$$\begin{aligned} \texttt{gather}_M \langle v \rangle_f\, s &\iff s = \circ \\ \texttt{gather}_M\, c\, s &\iff c \rhd c' \wedge \texttt{gather}_M\, c'\, s \\ \texttt{gather}_M\, c\,(d \cdot s) &\iff c \rhd^d c' \wedge \texttt{gather}_M\, c'\, s \end{aligned}$$

$$\texttt{eval}_M\, t\, s \iff \texttt{gather}_M \langle t \rangle_i\, s$$

The final equivalence result for this formalisation is stated as the following theorem:

Theorem 8.1. *For any term t and an evaluation trace s of a language satisfying axioms of Section 2.3.2, the equivalence* $\texttt{eval}\, t\, s \iff \texttt{eval}_M\, t\, s$ *holds.x*

Below we present the result of applying the transformation to the lambda calculus under call-by-value example of Section 4.1. The $\texttt{gather}_M$ and $\texttt{eval}_M$ relations should be understood as the largest relations satisfying the specified equivalences. Transitions of the machine are denoted with $\rhd$, with the ones emitting a decomposition d presented as $\rhd^d$.

$$\begin{aligned}
\langle t \rangle_i &\rhd \langle t, [\,] \rangle_e \\
\langle t_1\, t_2, E \rangle_e &\rhd \langle t_1, [\,]\, t_2 :: E \rangle_e \\
\langle v, E \rangle_e &\rhd \langle E, v \rangle_a \\
\langle [\,], v \rangle_a &\rhd^{v} \langle v \rangle_f \\
\langle [\,] t :: E, v \rangle_a &\rhd \langle t, v [\,] :: E \rangle_e \\
\langle (\lambda x.t) [\,] :: E, v \rangle_a &\rhd^{(\lambda x.t)\, v} \langle t[v/x], E \rangle_e
\end{aligned}$$

This formalisation is interesting not only because it lets us reason about non-terminating programs. It also gives us some additional insight into the nature of refocusing transformation: as the execution traces do not change during the transformation, we know that the resulting machine will contract the exact same redexes as the evaluator based on reduction semantics, and in the very same order. Additionally, with only a minor extensions, we could also reason about programs that get stuck – we would only need to extend the definition of `gather` by an equation for the stuck decompositions.

8.3 Coinduction in Coq

Coinduction in Coq is summarised in a very approachable and concise way in [26]. The principal method of both defining infinite data structures and proving theorems about them is the `cofix` operator, which allows for computing the greatest fixed point of a given set of rules. In the implementation of this formalisation we use tools based on this operator to specify the execution traces and these relations that are understood coinductively. This method is one of the newer additions to the proof assistant and as such it is still not very easy to use. However, as of now it is the only way to reason about such structures in Coq, and with some practice it can be an efficient tool.

In Figure 8.1 we present three fragments of the implementation of the formalisation: a generic definition of an eval-apply abstract machine's transition relation and the $\texttt{gather}_M$ and $\texttt{eval}_M$ functions used to define the semantics of abstract machines. The `dec_term` and `dec_context` functions used in the implementation are the $\texttt{dec}_t$ and $\texttt{dec}_f$ functions used throughout the formalisation. As we can see, the definitions do not differ much from the ones shown in Chapter 5 – beside some details specific to this formalisation, the only difference is replacing the `Inductive` keyword with `CoInductive`. The same analogy is visible in the proofs, though, as we have mentioned, the support for coinduction is not yet as well developed as the tools for inductive reasoning.

```
Inductive transition :
  configuration → configuration →
  option R.decomp → Prop :=
| t_init : forall t,
             ⟨t⟩_i ▷ ⟨t, []⟩_e ↑ None
| t_red  : forall t t0 E r,
             dec_term t = R.in_red r →
             R.contract r = Some t0 →
             ⟨t, E⟩_e ▷ ⟨t0, E⟩_e ↑ Some (R.d_red r E)
| t_val  : forall t v E,
             dec_term t = R.in_val v →
             ⟨t, E⟩_e ▷ ⟨E, v⟩_a ↑ None
| t_term : forall t t0 f E,
             dec_term t = R.in_term t0 f →
             ⟨t, E⟩_e ▷ ⟨t0, f::E⟩_e ↑ None
| t_cfin : forall v,
             ⟨[], v⟩_a ▷ ⟨v⟩_f ↑ Some (R.d_val v)
| t_cred : forall v t f E r,
             dec_context f v = R.in_red r →
             R.contract r = Some t →
             ⟨f::E, v⟩_a ▷ ⟨t, E⟩_e ↑ Some (R.d_red r E)
| t_cval : forall v v0 f E,
             dec_context f v = R.in_val v0 →
             ⟨f::E, v⟩_a ▷ ⟨E, v0⟩_a ↑ None
| t_cterm: forall v t f f0 E,
             dec_context f v = R.in_term t f0 →
             ⟨f::E, v⟩_a ▷ ⟨t, f0::E⟩_e ↑ None
where " c1 ▷ c2 ↑ d " := (transition c1 c2 d).

CoInductive gather_M : configuration → trace → Prop :=
| empty_trace : forall v,
             gather_M (c_final v) ∘
| empty_trans : forall c0 c1 s,
             c0 ▷ c1 ↑ None →
             gather_M c1 s →
             gather_M c0 s
| trace_trans : forall c0 c1 d s,
             c0 ▷ c1 ↑ Some d →
             gather_M c1 s →
             gather_M c0 (d · s).

CoInductive eval : term → trace → Prop :=
| e_intro : forall t ds,
             trace_m ⟨t⟩_i ds →
             eval t ds.
```

Figure 8.1: A Coq definition of an eval-apply abstract machine with evaluation function gathering an evaluation trace.

Chapter 9

Conclusions and perspectives

We have undertaken the effort of formalising the refocusing transformation. As is often the case, this work has not only led us to the answers, but also let us gain insight needed to identify new problems. In this chapter, we first summarise the results of the project and then proceed to identify possible directions of further development.

9.1 Conclusion

In Chapter 1 we have described our twofold approach to the problem of formalising the refocusing transformation. The first problem was the axiomatisation of reduction semantics in a way that allowed for applying the transformation automatically. This goal has been realised by providing the two sets of axioms, presented in Chapter 2. When considering the efforts of axiomatisation of a theory, there is always a question whether, on one hand, the formalisation provided is general enough to allow for many different realisations and, on the other hand, if it is powerful enough to derive all the required properties from it. By proving the transformation correct in Chapter 3 we have shown that the specification is strong enough to be used; the real difficulty, however, is whether it is actually useful, that is, can we show without a big effort, that a sophisticated language *does* conform to it. This is obviously not a property one could prove. However, by presenting the case studies of Chapter 4, we show that the formalisation can be easily used for languages of varying degrees of complexity, including some of the more sophisticated constructs, leading us to believe that the formalisation is successful.

The second part of our approach was to prove the whole refocusing transformation correct with respect to the provided specification. The work was carried out via a series of changes to the semantics, as described in Chapter 3. The proofs were rather straightforward, which we believe to be at least partially due to the axiomatisation used. We find the general and formal proof of correctness of the refocusing transformation to be one of the main contributions of this thesis.

Beside formalising the basic transformation, we have successfully managed to extend the formalisation to the calculi of closures, context sensitive reduction semantics and reasoning about nonterminating programs. This provides us with more arguments that the axiomatisation is chosen well, as the extensions fit it very well. Also, we note that especially the extension to context sensitive reduction semantics allow for even more sophisticated language constructs, such as control operators or op-

timised behaviour for languages with multiple binders. The extensions are fully orthogonal to each other, so one could combine them according to the needs. However, not every combination has been implemented, mostly due to the time constraints of the project.

The final aspect of the formalisation that we wish to mention here is the implementation in the Coq proof assistant. Thanks to the use of this tool the transformation can be used automatically, by simply plugging the formalisation of the language and its semantics into a correct functor. It also ensures the correctness of derivation, proofs of which get somewhat involved, especially when considering the extension to calculi of closures.

9.2 Future work

Although the project has finished with a rather conclusive results, during its course there have appeared certain ideas and perspectives for extensions that are summarised in this section.

As we have already stated, the extension to context sensitive reduction semantics allows us to refocus the languages with control operators, such as call/cc. However, more sophisticated control structures, for example the shift and reset operators, translate into 2CPS – a style of programming with two levels of continuations[11]. Thus, it could prove interesting to investigate reduction semantics exploiting two levels of reduction contexts and their connection to even more sophisticated constructs that appear in programming languages.

Another interesting possibility is applying refocusing to some lazy calculus, such that the machine obtained through the transformation would correspond to the STG machine of Haskell fame[21]. Such derivation could provide both an interesting lazy reduction semantics, which could give additional insight into lazy languages in general, and necessitate extensions to the transformation that could be interesting but hard to come up with otherwise. This direction can be connected with the previous one as the recent research of Danvy et al. shows[12].

There is also a number of small changes that could make the formalisation described in this thesis more elegant or easier to use. The formalisation of closure calculi is currently much harder to use then the substitution-based refocusing as well as the other extensions presented. Most probably, this is caused by the greater complexity of these calculi, but there might be a possibility to alleviate the complexity of formalisation. Also, some slight changes to formalisation might allow to improve code reuse in the implementation without sacrificing any generality or ease of use.

Bibliography

[1] Mads Sig Ager. From natural semantics to abstract machines. In Sandro Etalle, editor, *LOPSTR*, volume 3573 of *Lecture Notes in Computer Science*, pages 245–261. Springer, 2004.

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 8–19, New York, NY, USA, 2003. ACM.

[3] Malgorzata Biernacka and Olivier Danvy. Towards compatible and interderivable semantic specifications for the scheme programming language, part ii: Reduction semantics and abstract machines. In Jens Palsberg, editor, *Semantics and Algebraic Specification*, volume 5700 of *Lecture Notes in Computer Science*, pages 186–206. Springer, 2009.

[4] Małgorzata Biernacka and Dariusz Biernacki. Formalizing constructions of abstract machines for functional languages in coq. In J. Giesl, editor, *Preliminary Proceedings of the Seventh International Workshop on Reduction Strategies in Rewriting and Programming (WRS'07)*. Paris, France, 2007.

[5] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Logic*, 9(1):6, 2007.

[6] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theor. Comput. Sci.*, 375(1-3):76–108, 2007.

[7] Jacek Chrząszcz. Implementing modules in the Coq system. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286. Springer, 2003.

[8] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Sci. Comput. Program.*, 8(2):173–202, 1987.

[9] Pierre Crégut. An abstract machine for lambda-terms normalization. In *LISP and Functional Programming*, pages 333–340, 1990.

[10] Pierre-Louis Curien. An abstract framework for environment machines. *Theor. Comput. Sci.*, 82(2):389–402, 1991.

[11] Olivier Danvy and Andrzej Filinski. Representing control: A study of the cps transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[12] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. Defunctionalized interpreters for call-by-need evaluation. In Matthias Blume and German Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010*, number 6009 in Lecture Notes in Computer Science, pages 240–256, Sendai, Japan, April 2010. Springer.

[13] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Series RS-04-26, BRICS, Department of Computer Science, University of Aarhus, November 2004. iii+44 pp. This report supersedes BRICS report RS-02-04. A preliminary version appears in the informal proceedings of the *Second International Workshop on Rule-Based Programming*, RULE 2001, Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[14] Stephan Diehl. Natural semantics-directed generation of compilers and abstract machines. *Formal Asp. Comput.*, 12(2):71–99, 2000.

[15] Matthias Felleisen. *The calculi of lambda-nu-cs conversion: a syntactic theory of control and state in imperative higher-order programming languages*. PhD thesis, Indiana University, Indianapolis, IN, USA, 1987.

[16] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes.

[17] Matthias Felleisen and Daniel P. Friedman. Control operators, the secd-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, pages 193–219, Ebberup, Denmark, August 1986.

[18] Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 153–164. ACM, 2009.

[19] John Hannan and Dale Miller. From operational semantics to abstract machines. In *Mathematical Structures in Computer Science*, 1992.

[20] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *J. Funct. Program.*, 8(2):131–176, 1998.

[21] Simon L. Peyton Jones and Jon Salkild. The spineless tagless g-machine. In *FPCA*, pages 184–201, 1989.

[22] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[23] Jean-Louis Krivine. Un interpréteur du lambda-calcul, 1985. Brouillon. Unpublished.

[24] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

[25] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[26] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.

[27] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *J. Funct. Program.*, 16(4-5):415–449, 2006.

[28] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theor. Comput. Sci.*, 87(1):209–220, 1991.

[29] H. Riis Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

[30] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[31] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[32] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.