The Functional Correspondence Applied: An Implementation of a Semantics Transformer

(Zastosowanie odpowiedniości funkcyjnej
do implementacji transformatora semantyk)

Maciej Buszka

Praca magisterska

Promotor: dr hab. Dariusz Biernacki

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

11 lipca 2020

Abstract

I transform a manual derivation technique known as the functional correspondence between evaluators and abstract machines into a robust algorithm.

I begin by describing the classic formulation of the methodology consisting of two source-to-source human-aided transformations and characterize control-flow analysis as a basis for their algorithmization. I then define the meta-language *IDL* and present the three main steps of the automatic procedure: transformation to administrative normal form, selective transformation to continuation-passing style and selective defunctionalization. I also derive a procedure for computing control-flow analysis of programs in *IDL* using abstracting abstract machines methodology.

The thesis is accompanied by an implementation of the algorithm in the form of a command-line tool. It allows for transformation of an interpreter embedded in a *Racket* source file and gives fine-grained control over the resulting machine. I present a selection of case-studies which showcase the performance of the tool and the algorithm by deriving both known and novel abstract machines.

W pracy pokazuję jak przekształcić ręczną metodę derywacji, znaną jako odpowiedniość funkcyjna pomiędzy ewaluatorami i maszynami abstrakcyjnymi, w uniwersalny algorytm.

Zaczynając od klasycznego sformułowania metodologii, składającego się z dwóch transformacji kodu źródłowego wymagających pracy człowieka, wskazuję na analizę przepływu sterowania jako podstawę do ich algorytmizacji. Następnie definiuję meta-język *IDL* i przedstawiam trzy główne etapy automatycznej procedury: transformację do adminstracyjnej postaci normalnej, wybiórczą transformację do stylu przekazywania kontynuacji oraz wybiórczą defunkcjonalizację. Pokazuję także procedurę pozwalającą na obliczenie analizy przepływu sterowania dla programów w *IDL*, którą otrzymałem stosując metodologię abstrahowania maszyn abstrakcyjnych.

Do pracy dołączona jest implementacja algorytmu w postaci programu używanego z wiersza poleceń. Pozwala ona na transformację interpretera zanużonego w pliku źródłowym w języku *Racket* oraz zapewnia precyzyjną kontrolę nad kształtem wynikowej maszyny. W pracy przedstawiam zbiór przykładowych interpreterów na których obrazuję działanie narzędzia i algorytmu poprzez derywację zarówno znanych jak i nowych maszyn abstrakcyjnych.

Contents

Chapter 1

Introduction

What is the meaning of a given computer program?

The field of formal semantics of programming languages seeks to provide tools to answer such a question. Denotational semantics [1] allow one to relate programs to mathematical objects which describe their behavior. Operational semantics provide means to characterize evaluation of programs by building a relation between terms and final values in case of natural (aka big-step) semantics [2] and pretty-big-step semantics [3]; by defining a step-by-step transition system on program terms in case of structured operational (aka small-step) semantics [4] and reduction semantics [5]; or by specifying an abstract machine with a set of states and a transition relation between those states. All of these semantic formats enable systematic definition of programming languages but differ in style and type of reasoning they allow as well as in their limitations.

Diversity of formats of operational semantics and trade-offs they impose often necessitates specifying the semantics of a calculus in more than one format, e.g., one might use natural semantics in order to show a program transformation correct but will have to also specify small-step operational semantics for proofs of type safety and characterization of non-terminating computations. Of course when multiple specifications are provided one should also prove them compatible. As it implies serious amount of work, it comes as no surprise that research has been conducted on means of mechanizing or even automating this task. For instance, in their paper [6] Poulsen and Mosses show an automatic procedure for obtaining pretty-big-step semantics from small-step ones. The most recent result is a 2019 paper by Vesely and Fisher [7] who describe an automatic transformation in other direction: from big-step semantics into its small-step counterpart.

Another line of work is that of constructing abstract machines. Starting with Landin's SECD machine [8] for λ-calculus, many abstract machines have been proposed for various evaluation strategies and with differing assumptions on capabilities of the runtime (e.g., substitution vs environments). Notable work includes: Kriv-

ine's machine [9] for call-by-name reduction, Felleisen and Friedman's CEK machine [10] and Crégut's machine [11] for normalization of λ-terms in normal order. Besides equipping existing calculi with an abstract machine, new developments also come with both higher-level operational semantics and a machine, e.g., in the novel field of algebraic effects [12, 13]. Manual construction of an abstract machine for a given evaluation discipline can be challenging and also requires a proof of compatibility w.r.t the source semantics, therefore methods for deriving the machines have been developed. Danvy and Nielsen's refocusing framework [14] gave raise to an automatic procedure for transforming reduction semantics into an abstract machine [15, 16]. Ager shows a mechanical method of deriving abstract machines from L-attributed natural semantics [17] while Hannan and Miller present derivations of abstract machines for call-by-value and call-by-name reduction strategies [18] via program transformations. Last but not least, Danvy et al. introduced the functional correspondence between evaluators and abstract machines [19] which appears to be the most successful technique.

In order to describe the functional correspondence in greater detail let us first turn to another approach to defining a programming language: providing an interpreter for the language in question (which I will call the *object*-language) written in another language (to which I will refer as the *meta*-language). These definitional interpreters [20] can be placed on a spectrum from most abstract to most explicit. At the abstract end lie the concise meta-circular interpreters which use meta-language constructs to interpret same constructs in object-language (e.g., using anonymous functions to model functional values, using conditionals for *if* expressions, etc.). In the middle one might place various evaluators with some constructs interpreted by simpler language features (e.g., with environments represented as lists or dictionaries instead of functions) but still relying on the evaluation order of the meta-language. The explicit end is occupied by first-order machine-like interpreters which use an encoding of a stack for handling control-flow of the object-language.

In his seminal paper [20] Reynolds introduces two techniques: transformation to continuation-passing style and defunctionalization, which allow one to transform high-level definitional interpreters into lower-level ones. This connection between evaluators on different levels of abstraction has later been studied by Danvy et al.[19] who use it to relate several abstract machines for λ-calculus with interpreters embodying their evaluation strategies and called it the functional correspondence. The technique has proven to be very useful for deriving a correct-by-construction abstract machine given an evaluator in a diverse set of languages and calculi including normal and applicative order λ-calculus evaluation [19] and normalization [21], call-by-need strategy [22] and *Haskell*'s STG language [23], logic engine [24], delimited control [25], computational effects [26], object-oriented calculi [27] and *Coq*'s tactic language [28]. Besides the breadth of applications the functional correspondence proved able to relate semantic formats from opposing ends of the abstraction spectrum, e.g., a meta-circular interpreter encoding call-by-value denotational semantics for λ-calculus

with CEK machine or a normal order normalization function with a strong version of Krivine's machine [21]. Despite these successes and its mechanical nature, the functional correspondence has not yet been transformed into a working tool which would perform the derivation automatically.

Therefore, it was my goal to give an algorithmic presentation of the functional correspondence and implement this algorithm in order to build a semantics transformer. In this thesis I describe all steps required to successfully convert the human-aided derivation into a computer algorithm for transforming evaluators into a representation of an abstract machine. In particular I characterize the control-flow analysis as the basis for both selective continuation-passing style transformation and partial defunctionalization. In order to obtain correct, useful and computable analysis I employ the abstracting abstract machines methodology (AAM) [29] which allows for deriving the analysis from an abstract machine for the meta-language. This derivation proved very capable in handling the non-trivial language containing records, anonymous functions and pattern matching. The resulting analysis enables automatic transformation of user specified parts of the interpreter as opposed to whole-program-only transformations. I implemented the algorithm in the *Haskell* programming language giving raise to a tool — `semt` — performing the transformation. I evaluated the performance of the tool on multiple interpreters for a diverse set of programming language calculi.

The two transformations which form the functional correspondence have been studied and proven correct in various settings. The transformation to continuation-passing style is often used in the context of compilation of programming languages [30]. In particular selective variants of the transformation have been proposed and proven in context of control operators [31]. The main approach to distinguishing terms which should be transformed is to provide a type system annotated with required information. The defunctionalization can also be used as a compilation technique but it has not seen as much use as other transformations such as closure conversion. Nevertheless there are formulations of defunctionalization based on control-flow analysis results [32] which were proven correct. As with the CPS transformation, the information is also embedded in a type system. I chose to base both transformations on a separately computed control-flow analysis as it allowed me to choose the analysis which best suited the meta-language. By using the AAM methodology I obtained results enabling powerful transformations without the implementation complexity of type inference, requiring annotations in the source program or turning two type systems into constraint-based analyses.

The rest of this thesis is structured as follows: In the remainder of this chapter I introduce the *Interpreter Definition Language* which is the meta-language accepted by the transformer and will be used in example evaluators throughout the thesis; I also compare the semantics formats with styles of interpreters to which they correspond. In Chapter 2, I describe the functional correspondence and its constituents. In Chapter 3, I show the algorithmic characterization of the correspondence. In

Chapter 4, I showcase the performance of the tool on a selection of case studies. In Chapter 5, I discuss related work, point at future avenues for improvement and conclude. Appendix A contains user's manual for the semantic transformer.

I assume that the reader is familiar with λ-calculus and its semantics (both normal (call-by-name) and applicative (call-by-value) order reduction). Familiarity with formal semantics of programming languages (both denotational and operational) is also assumed although not strictly required for understanding of the main subject of this thesis. The reader should also be experienced in using a higher-order functional language with pattern matching.

1.1 Interpreter Definition Language

The *Interpreter Definition Language* or *IDL* is the meta-language used by `semt` – a semantic transformer. It is a purely functional, higher-order, dynamically (strongly) typed language with strict evaluation order. It features named records and pattern matching which allow for convenient modelling of abstract syntax of the object-language as well as base types of integers, booleans and strings. The concrete syntax is in fully parenthesized form and the programs can be embedded in a `Racket` source file using the provided library with syntax definitions. A more detailed introduction along with usage instructions is available in Appendix A.

As shown in Figure 1.1 a typical interpreter definition consists of several top-level functions which may be mutually recursive. The `def-data` form introduces a datatype definition. In our case it defines a type for terms of λ-calculus – `Term`. It is a union of three types: `String`s representing variables of λ-calculus; records with label `Abs` and two fields of types `String` and `Term` representing abstractions; and records labeled `App` which contain two `Term`s and represent applications. A datatype definition may refer to itself, other previously defined datatypes and records and the base types of `String`, `Integer`, `Boolean` and `Any`. The `main` function is treated as an entry point for the evaluator and must have its arguments annotated with their type.

The `match` expression matches an expression against a list of patterns. Patterns may be variables (which will be bound to the value being matched), wildcards `_`, base type patterns, e.g., `[String x]` or record patterns, e.g., `{Abs x body}`. The `fun` form introduces anonymous function, `error "..."` stops execution and signals the error. Finally, application of a function is written as in *Scheme*, i.e., as a list of expressions (e.g., `(eval init term)`).

1.2 Semantic Formats

In this thesis I consider three widely recognized semantic formats: denotational semantics, big-step operational semantics and abstract machines. These formats

```
(def-data Term
  String
  {Abs String Term}
  {App Term Term})

(def init (x) (error "empty environment"))

(def extend (env y v)
  (fun (x) (if (eq? x y) v (env x))))

(def eval (env term)
  (match e
    ([String x] (env x))
    ({Abs x body} (fun (v) (eval (extend env x v) body)))
    ({App fn arg} ((eval env fn) (eval env arg)))))

(def main ([Term term]) (eval init term))
```

Figure 1.1: A meta-circular interpreter for λ-calculus

make different trade-offs with respect to conciseness of definition, explicitness of specification of behavior of the object-language and power or degree of complication of the meta-language. I assume familiarity with these formats and the rest of this section should be treated as a reminder rather than an introduction. Nevertheless I will explain how these mathematical formalisms correspond to evaluators in a functional programming language.

Denotational Semantics

In this format one has to define a mapping from program terms into meta-language objects (usually functions) which *denote* those terms – that is they specify their behavior [1]. This mapping is usually required to be compositional – i.e., the denotation of complex term is a composition of denotations of its sub-terms. Denotational semantics are considered to be the most abstract way to specify behavior of programs and can lead to very concise definitions. The drawback is that interesting language features such as loops and recursion require more complex mathematical theories to describe the denotations, in particular domain theory and continuous functions. In terms of interpreters, the denotational semantics usually correspond to evaluators that heavily reuse features of the meta-language in order to define the same features of object-language, e.g., using anonymous functions to model functional values, using conditionals for *if* expressions, etc. This style of interpreters is sometimes called *meta-circular* due to the recursive nature of the language definition. On the

one hand these definitional interpreters allow for intuitive understanding of object-language's semantics given familiarity with meta-language. On the other hand, the formal connection of such an interpreter with the denotational semantics requires formal definition of meta-language and in particular understanding of the domain in which denotations of meta-language programs live. The evaluator of Figure 1.1 is an example of the meta-circular approach. The λ-abstractions of object-language are represented directly as functions in meta-language which use denotations of lambda's bodies in extended environment. The `eval` function is compositional – the denotation of object level application is an application of denotations of function and argument expressions.

Big-step Operational Semantics

The format of big-step operational semantics [2], also known as natural semantics allows for specification of behavior of programs using inference rules. These rules usually decompose terms syntactically and give rise to a relation between programs and values to which they evaluate. The fact of evaluation of a program to a value is proven by showing a derivation tree built using the inference rules. Non-terminating programs therefore have no derivation tree which makes this semantic format ill-suited to describing divergent or infinite computations. The interpreters which correspond to big-step operational semantics usually have a form of recursive functions that are not necessarily compositional. The natural semantics may be non-deterministic and relate a program with many results. When turning nondeterministic semantics into an evaluator (in a deterministic programming language) one has to either change the formal semantics or model the nondeterminism explicitly. Let us now turn to a simple interpreter embodying the natural semantics for an imperative language *IMP* shown in Figure 1.2.

Datatypes `AExpr`, `BExpr` and `Cmd` describe abstract syntax of arithmetic expressions, boolean expressions and commands. The expressions are pure, that is, evaluating them does not affect the state. The state is a function mapping variables represented as `String`s to numbers, initially set to `0` for every variable. Functions `aval` and `bval` valuate arithmetic and boolean expressions in a given state. The function `eval` is a direct translation of big-step operational semantics for *IMP*. It is not compositional in the `While` branch, where `eval` is called recursively on the same command it received.

Abstract Machines

An abstract machine [8] is usually the most explicit definition of semantics of a language with all the details like argument evaluation order, term decomposition, environments and closures specified. It is a format of particular interest as it can very precisely specify the operational properties of the language. Therefore it provides a

```
(def-data AExpr
  String
  ...)
(def-data BExpr ...)
(def-data Cmd
  {Skip}
  {Assign String AExpr}
  {If BExpr Cmd Cmd}
  {Seq Cmd Cmd}
  {While BExpr Cmd})

(def init-state (var) 0)
(def update-state (tgt val state) ...)

(def aval (state aexpr) ...) ;; valuate arithmetic expression
(def bval (state bexpr) ...) ;; valuate boolean expression

(def eval (state cmd)
  (match cmd
    ({Skip} state)
    ({Assign var aexpr}
      (update-state var (aval state aexpr) state))
    ({If cond then else}
      (if (bval state cond)
        (eval state then)
        (eval state else)))
    ({Seq cmd1 cmd2}
      (let state (eval state cmd1))
      (eval state cmd2))
    ({While cond cmd}
      (if (bval state cond)
        (eval (eval state cmd) {While cond cmd})
        state))))

(def main ([Cmd cmd])
  (eval init-state cmd))
```

Figure 1.2: An interpreter for *IMP* in the style of natural semantics

reasonable cost model of the evaluation and may even serve as a basis of efficient implementation [33].

A machine consists of a set of configurations (tuples), an injection of a pro-

gram into the initial configuration, an extraction function of a result from the final configuration and a transition relation between configurations. The behavior of the machine then determines the behavior of the programs in object-language. As with big-step operational semantics, an abstract machine may be nondeterministic. Usually elements of the machine-state tuple are simple and first-order, e.g., terms of the object-language, numbers, lists, etc. One way of encoding a deterministic abstract machine in a programming language is to define a function for each subset of machine states with similar structure. The exact configuration is determined by the actual parameters of the function at run-time. The bodies of these (mutually recursive) functions encode the transition function.

Figure 1.3 contains an interpreter corresponding to Krivine's machine [9] performing normal order (call-by-name) reduction of λ-calculus with de Bruijn indices. It uses two stacks: `Continuation` and `Environment`. Both of them contain `Thunks` – not-yet-evaluated terms paired with their environment. The object-language functions are represented as `Closures` – function bodies paired with their environment. The machine has two classes of states: `eval` and `continue`. The initial configuration is `(eval term {Nil}{Halt})` – i.e., eval with the term of interest and empty stacks. There are four transitions from `eval` configuration. The first two search for the `Thunk` corresponding to the variable (de Bruijn index) in the environment and then evaluate it with old stack but with restored environment. The third transition switches to `continue` configuration with the closure is created by pairing current environment with abstraction's body. The fourth transition pushes a `Thunk` onto continuation stack and begins evaluation of function expression. In the `continue` configuration the machine inspects the stack. If it is empty then the computed function `fn` is the final answer which is returned. Otherwise an argument is popped from the stack and the machine switches to `eval`uating the `body` of the function in the restored environment extended with `arg`.

```
(def-data Term
  Integer
  {Abs Term}
  {App Term Term})

(def-struct {Closure body env})
(def-struct {Thunk env term})

(def-data Env
  {Nil}
  {Cons Thunk Env})

(def-data Cont
  {Push Thunk Cont}
  {Halt})

(def eval ([Term term] [Env env] cont)
  (match term
    (0 (match env
         ({Nil} (error "empty env"))
         ({Cons {Thunk env term} _} (eval term env cont))))
    ([Integer n] (eval (- n 1) env cont))
    ({Abs body} (continue cont {Closure body env}))
    ({App fn arg} (eval fn env {Push {Thunk env arg}}))))

(def continue (cont fn)
  (match cont
    ({Push arg cont}
      (let {Closure body env} fn)
      (eval body {Cons arg env} cont))
    ({Halt} fn)))

(def main ([Term term]) (eval term {Nil} {Halt}))
```

Figure 1.3: An encoding of Krivine's machine

Chapter 2

The Functional Correspondence

The functional correspondence between evaluators and abstract machines is a technique for mechanical derivation of an abstract machine from a given evaluator.[1] The technique was first characterized and described in [19] and then later studied in context of various object-languages and their evaluators in [21, 22, 23, 24, 25, 26, 27, 28]. The input of the derivation is an evaluator written in some functional meta-language. It usually corresponds to a variant of denotational semantics (particularly in case of so-called meta-circular interpreters) or big-step operational semantics. The result of the derivation is a collection of mutually tail-recursive, first-order functions in the same meta-language. Program in such a form corresponds to an abstract machine. The different functions (with actual parameters) represent states of the machine, while the function calls specify the transition function.

The derivation consists of two program (in our case interpreter) transformations: transformation to continuation-passing style and defunctionalization. The first one exposes the control structure of the evaluator; the second replaces function values with first-order data structures and their applications with calls to a first-order global function.

In the remainder of this chapter I will describe those transformations and illustrate their behavior using the running example of an evaluator for λ-calculus from Section 1.1. Let us recall the previously described meta-circular interpreter of Figure 1.1. The variables are represented as `String`s of characters. Since every closed expression in λ-calculus may only evaluate to a function, the values produced by the interpreter are represented as meta-language functions. The interpreter uses environments represented as partial functions from variables to values to handle binding of values to variables during application. The application in the object-language is interpreted using application in the meta-language, so the defined language inherits call-by-value, left-to-right evaluation order. At the end of this chapter we shall arrive at the CEK machine.

[1] It may also be used in the other direction, i.e., to derive an interpreter corresponding to an abstract machine, by using inverse transformations.

2.1 Continuation-Passing Style

The first step towards building the abstract machine is capturing the control-flow characteristics of the defined language. We are interested in exposing the order in which the sub-expressions are evaluated and how the control is passed from function to function. Additionally, we would like the resulting program to define a transition system so we must require that every function call in the interpreter is a tail-call. It turns out that a program in continuation-passing style (CPS) exactly fits our requirements.

What does it mean for a program to be in CPS? Let us begin by classifying expressions into trivial and serious ones. An expression is trivial if evaluating it always returns a value. Since we cannot in general decide whether an arbitrary expression is trivial we will use a safe approximation: an expression is trivial if it is a variable, a function definition, a primitive operation call or a structure constructor with only trivial expressions as sub-terms. We will only allow applications, match expressions and constructors with trivial sub-expressions. Additionally we would like to consider some expressions trivial due to their interpretation (e.g., environment lookups) even though they are serious. In order to retain ability to build interesting programs, every function will accept an additional argument – a continuation which specifies what should be done next.

The interesting clauses of the algorithm for simple CPS translation of expressions in *IDL* are presented in Figure 2.1. The meta variables are typeset with italics (e.g., k). The pieces of syntax use typewriter font (e.g., `k'`). The function $[\![e]\!]k$ transforms an expression e to continuation-passing-style using expression k as a continuation. Whenever a new variable is introduced by the algorithm we will assume that it is fresh. The variable `x` is translated to application of continuation k to `x`. To translate an anonymous function definition, first a fresh variable `k'` is generated then the body of the function is translated with `k'` as the continuation and finally, the continuation k is applied to the transformed function expression. Function application is transformed by placing all sub-expressions in successively nested functions, with the deepest one actually performing the call with an additional argument – the continuation k. This way the evaluation of arguments is sequenced left-to-right and happens before the application. Translation of the `match` expression requires translating the scrutinee and putting the branches in the continuation. The branches are all transformed using the same continuation k. Finally, during translation of the `error` expression the continuation is discarded since the error halts the execution. The omitted rules for `if` expressions and record creation are similar to `match` expressions and applications, respectively.

Figure 2.2 shows the interpreter with body of `eval` translated to CPS using the algorithm of Figure 2.1 and then hand-optimized by reducing administrative redexes. The `eval` function now takes an additional argument `k` – a continuation. The function

$$
\begin{aligned}
[\![\texttt{x}]\!]k &= (k\ \texttt{x}) \\
[\![(\texttt{fun}\ (x \ldots)\ e)]\!]k &= (k\ (\texttt{fun}\ (x \ldots \texttt{k'})\ [\![\texttt{e}]\!]\texttt{k'})) \\
[\![(e_1 \ldots e_n)]\!]k &= [\![e_1]\!]\,(\texttt{fun}\ (\texttt{v}_1)\ [\![e_2]\!]\,(\texttt{fun}\ (v_2) \ldots [\![e_n]\!]k' \ldots) \\
&\quad \text{where } k' = (\texttt{fun}\ (\texttt{v}_n)\ (\texttt{v}_1 \ldots \texttt{v}_n\ k)) \\
[\![(\texttt{match}\ e\ (p\ e) \ldots)]\!]k &= [\![e]\!](\texttt{fun}\ (\texttt{v})\ (\texttt{match}\ \texttt{v}\ ps) \\
&\quad \text{where } ps = (p\ [\![e']\!]k) \ldots
\end{aligned}
$$

Figure 2.1: A call-by-value CPS translation

denoting the object-language lambda expressions also expects a continuation. In both variable and abstraction cases the evaluator now calls the continuation `k` with the computed value: either looked up in the environment in case of a variable or freshly constructed in case of abstractions. The evaluation of applications is now explicitly sequenced. First the expression in function position will be evaluated. It is passed a continuation which will then evaluate the argument. After the argument is computed, the function value will be applied to the argument and the original continuation `k` passed by the caller. The `main` function is kept in direct style as it is the entry point of the evaluator. It calls the `eval` function which expects a continuation so it provides it the identity function. This continuation means that when evaluation is finished it will return the final value.

We can see that after the transformation the evaluation order of the meta-language does not affect the evaluation order of the object-language as every call to the only interesting function `eval` is a tail call. Therefore we have successfully captured control-flow characteristics of the object-language. The evaluator still technically depends on the order of evaluation of *IDL* as environment lookup may fail and it is in a sub-expression position but from the point of view of designing an abstract machine which works with closed terms it is not interesting.

In Section 3.3 we will see a more complex transformation which avoids creating administrative redexes and allows for user defined functions which should be considered trivial.

2.2 Defunctionalization

The second step is the elimination of higher order functions from our interpreter, transforming it into a collection of mutually (tail-)recursive functions – a state machine with the `main` function building initial configuration. There are many approaches to compiling first class functions away but of particular interest to us will

```
(def-data Term ...)

(def eval (env term k)
  (match term
    ([String x] (k (env x)))
    ({Abs x body}
      (k (fun (v k') (eval (extend env x v) body k'))))
    ({App fn arg}
      (eval env fn
        (fun (fn') (eval env arg (fun (v) (fn' v k))))))))

(def extend (env x v) ...)

(def init (x) (error "empty environment"))

(def main ([Term term]) (eval init term (fun (x) x)))
```

Figure 2.2: An interpreter for λ-calculus in CPS

be defunctionalization. It is a global program transformation that replaces each anonymous function definition with a uniquely labeled record which holds the values for function's free variables. Every application of unknown function is replaced with a specific top-level *apply* function which dispatches on the label of the passed record and evaluates the corresponding function's body.

This simple description glosses over many important details. Firstly, we must distinguish between known and unknown function calls as only unknown calls should be transformed. Secondly, we must be able to create records for top-level definitions when they are passed as a first-class function, e.g., in the definition of `eval` in the branch for variables we apply an unknown function `env` which may evaluate either to an anonymous function created by `extend` or a top-level function `init`. Lastly, we must somehow know for each application point which functions may be applied. The first two challenges can be solved with a static, syntactic analysis of the interpreter. The other challenge can be solved using control-flow analysis as described in Section 3.4. For the purposes of this example we observe that there are three function spaces with anonymous functions: continuations, representation of abstractions and environments.

Figure 2.3 depicts an overview of defunctionalization procedure with *apply* functions generated according to the template in Figure 2.4. We assume that every definition and expression in a program has a unique label and that all generated names and structure labels are fresh. Whenever a top-level function is called the application is transformed into top-level call with the sub-expressions transformed. Any other

$$
\begin{aligned}
[\![(\texttt{f}\ e_1 \ldots e_n)\texttt{@l}]\!] &= (\texttt{f}\ [\![e_1]\!] \ldots [\![e_n]\!]) && \text{when f is top-level} \\
[\![(\ e_1 \ldots e_n)\texttt{@l}]\!] &= (\texttt{apply-l}\ [\![e_1]\!] \ldots [\![e_n]\!]) && \text{otherwise} \\
[\![(\texttt{fun (x ...)}\ e_l)\texttt{@l}]\!] &= \{\texttt{l y ...}\} \\
[\![\texttt{f}]\!] &= \{\texttt{lf}\}
\end{aligned}
$$

Figure 2.3: Defunctionalization algorithm

```
(def apply-l (f x ...)
  (case f
    ({l-1 y-1 ...} e-1)
    ...
    ({l-n y-n ...} e-n)))
```

Figure 2.4: Apply function template

application is transformed into a call to `apply-l` where `l` is the application's label. Anonymous functions are transformed into a labeled record with the function's free variables as sub-expressions. Finally, references to top-level functions occurring in the program are transformed into labeled records. The template in Figure 2.4 is instantiated as follows:

- `l` is a label of application expression for which `apply-l` is generated
- `l-1` ... `l-n` are labels of functions which may be applied in `l`
- `x ...` are variables bound by these functions (notice that it requires renaming of bound variables)
- (`y-1 ...`) ... (`y-n ...`) are free variables of these functions
- `e-1` ... `e-n` are already transformed bodies of these functions

It is worth noting that defunctionalization preserves the tail-call property of a program in CPS. After applying the defunctionalization procedure to functions representing lambda abstractions and to continuations we obtain (again with a bit of manual cleanup) an encoding of the CEK machine [34] in Figure 2.5. It uses a stack `Cont` (which are defunctionalized continuations) to handle the control-flow and `Closures` (which are defunctionalized lambda abstractions) to represent functions. The environment is left untouched and is still encoded as a partial function. The machine has two classes of states: `eval` and `continue`. In `eval` mode the machine dispatches on the shape of the term and either switches to `continue` mode when it has found a value (either a variable looked up in the environment or an abstraction)

or pushes a new continuation onto the stack and evaluates the expression in function position. The `continue` function is the *apply* function generated by the defunctionalization procedure. In `continue` mode the machine checks the continuation and proceeds accordingly: when it reaches the bottom of the continuation stack `Halt` it returns the final value `val`; when the continuation is `App1` it means that `val` holds the function value which will be applied once `arg` is computed; the stack frame `App2` signifies that `val` holds the computed argument and the machine calls a helper function `apply` (the second generated *apply* function) to unpack the closure in `fn` and evaluate the body of the closure in the extended environment.

```
(def-data Term ...)

(def-data Cont
  {Halt}
  {App1 arg env cont}
  {App2 fn cont})

(def-struct {Closure body env x})

(def init (x) (error "empty environment"))
(def extend (env k v) ...)

(def eval (env term cont)
  (match term
    ([String x] (continue cont (env x)))
    ({Abs x body} (continue cont {Closure body env x}))
    ({App fn arg} (eval env fn {App1 arg env cont}))))

(def apply (fn v cont)
  (let {Fun body env x} fn)
  (eval (extend env x v) body cont))

(def continue (cont val)
  (match cont
    ({Halt} val))
    ({App1 arg env cont} (eval env arg {App2 val cont}))
    ({App2 fn cont} (apply fn val cont)))

(def main ([Term term]) (eval {Init} term {Halt}))
```

Figure 2.5: An encoding of the CEK machine for λ-calculus

Chapter 3

Semantics Transformer

The adaptation of the functional correspondence into a semantics transformer was the main goal of this thesis. Although the two main transformations considered here are widely known, they are not presented in literature in a form directly applicable for the task. As the goal of the algorithm is to produce a definition of an abstract machine, to be read as a source code, care has to be taken to produce readable results. To this end I chose to allow for partial CPS translation, with functions which should be left alone marked with annotations. This approach allows one to specify helper functions whose control flow is not particularly interesting to capture such as environment lookups. The defunctionalization is usually presented as a manual transformation with human-specified function spaces or in a type directed fashion. Neither of these approaches are satisfying for the purposes of a semantics transformer as the goal is to produce the result automatically and to uncover the operational properties of the evaluator. Additionally I wanted to allow for partial defunctionalization of programs as it permits one to keep some parts of the machine abstract (e.g., functions modelling a heap or an environment). I chose to employ the control-flow analysis to guide partitioning of function spaces as it approximates the runtime behavior of programs. This approach allows for functions of the same type to land in different spaces based on their usage and it performed satisfactorily in experiments I have conducted. Finally, as the transformation generates new variables and moves code around we have to keep them readable. To this end I both allow for (optional) program annotations and employ heuristics to guide generation of names for function records, introduced variables and functions.

The abstract syntax of *IDL* is presented in Figure 3.1. The meta-variables x, y, z denote variables; r denotes structure (aka record) names; s is used to denote string literals and b is used for all literal values – strings, integers and booleans. The meta-variable tp is used in pattern matches which check whether a value is one of the primitive types. The patterns are referred to with variable p and may be a variable, a literal value, a wildcard, a record pattern or a type test. Terms are denoted with variable t and are either a variable, a literal value, an anonymous function, an

$$
\begin{array}{rcl}
x, y, z \in Var & r \in StructName \quad s \in String & b \in Int \cup Boolean \cup String \\
Tp \ni tp & ::= & \texttt{String} \mid \texttt{Integer} \mid \texttt{Boolean} \\
Pattern \ni p & ::= & x \mid b \mid \_ \mid \{r\ p \ldots\} \mid [tp\ x] \\
Term \ni t & ::= & x \mid b \mid (\texttt{fun}\ (x \ldots)\ t) \mid (t\ t \ldots) \mid \{r\ t \ldots\} \\
 & & \mid (\texttt{let}\ p\ t\ t) \mid (\texttt{match}\ t\ (p\ t) \ldots) \mid (\texttt{error}\ s) \\
FunDef \ni fd & ::= & (\texttt{def}\ x\ (x \ldots)\ t) \\
StructDef \ni sd & ::= & (\texttt{def-struct}\ \{r\ x \ldots\})
\end{array}
$$

Figure 3.1: Abstract syntax of *IDL*

application, a record constructor, a let binding (which may destructure bound term with a pattern), a pattern match or an error expression.

The transformation described in this chapter consists of three main stages: translation to administrative normal form, selective translation to continuation-passing style and selective defunctionalization. After defunctionalization the program is in the desired form of an abstract machine. The last step taken by the transformer is inlining of administrative let-bindings introduced by previous steps in order to obtain more readable results. In the remainder of this chapter I will describe the three main stages of the transformation and the algorithm used to compute the control-flow analysis.

3.1 Administrative Normal Form

The administrative normal form (ANF) [35] is an intermediate representation for functional languages in which all intermediate results are let-bound to names. This shape greatly simplifies later transformations as programs do not have complicated sub-expressions. From the operational point of view, the only place where a continuation is grown when evaluating program in ANF is a let-binding. This property ensures that a program in ANF is also much easier to evaluate using an abstract machine which will be taken advantage of in Section 3.2. The abstract syntax of terms in ANF and an algorithm for transforming *IDL* programs into such form is presented in Figure 3.2. The terms are partitioned into three levels: variables, commands and expressions. Commands c extend variables with values – base literals, record constructors (with variables as sub-terms) and abstractions (whose bodies are in ANF); and with redexes like applications of variables and match expressions (which match on variable and have branches in ANF). Expressions e in ANF have the shape of a possibly empty sequence of let-bindings ending with either an error term or a command.

The $[\![\cdot]\!]\,\cdot$ function, written in CPS, is the main transformation function. Its arguments are term to be transformed and a meta-language continuation which will be called to obtain the term for the rest of the transformed input. This function

$$
\begin{array}{rl}
\mathit{Command} \ni c & ::= x \mid b \mid (\texttt{fun}\ (x \ldots)\ e) \mid (x\ x \ldots) \\
 & \mid \{r\ x \ldots\} \mid (\texttt{match}\ x\ (p\ e) \ldots) \\
\mathit{Expression} \ni e & ::= c \mid (\texttt{let}\ p\ c\ e) \mid (\texttt{error}\ s)
\end{array}
$$

$$
\begin{array}{rl}
[\![\cdot]\!]\ \cdot & : \mathit{Expr} \times (\mathit{Com} \to \mathit{Anf}) \to \mathit{Anf} \\
[\![a]\!]\ k & = k\, a \\
[\![(\texttt{fun}\ (x \ldots) e)]\!]\ k & = k\, (\texttt{fun}\ (x \ldots)\ [\![e]\!]\, id) \\
[\![(e_f\ e_{arg} \ldots)]\!]\ k & = [\![e_f]\!]\, [\lambda a_f.\, [\![e_{arg} \ldots]\!]_s\, \lambda(a_{arg} \ldots).k\, (a_f\ a_{arg} \ldots)]_a \\
[\![(\texttt{let}\ x\ e_1\ e_2)]\!]\ k & = [\![e_1]\!]\, \lambda e_1'.(\texttt{let}\ x\ e_1\ [\![e_2]\!]\, k) \\
[\![\{r\ e \ldots\}]\!]\ k & = [\![e \ldots]\!]_s\, \lambda(a \ldots).k\, \{r\ a \ldots\} \\
[\![(\texttt{match}\ e (p\ e_b))]\!]\ k & = [\![e]\!]\, [\lambda e'.k\, (\texttt{match}\ e\ (p\ [\![e_b]\!]\, id)]_a \\
[\![(\texttt{error}\ s)]\!]\ \_ & = (\texttt{error}\ s)
\end{array}
$$

$$
\begin{array}{rl}
[\cdot]_a\ \cdot & : (\mathit{Atomic} \to \mathit{Anf}) \to \mathit{Com} \to \mathit{Anf} \\
[k]_a\ a & = k\, a \\
[k]_a\ c & = (\texttt{let}\ x\ c\ (k\, x))
\end{array}
$$

$$
\begin{array}{rl}
[\![\cdot]\!]_s\ \cdot & : \mathit{Expr}^* \times (\mathit{Atomic}^* \to \mathit{Anf}) \to \mathit{Anf} \\
[\![e \ldots]\!]_s\ k & = [\![e \ldots]\!]_s^{\epsilon} \\
[\![\epsilon]\!]_s^{a \ldots}\ k & = k\, (a \ldots) \\
[\![e\, e_r \ldots]\!]_s^{a_{acc} \ldots}\ k & = [\![e]\!]\, [\lambda a.\, [\![e_r \ldots]\!]_s^{a_{acc} \ldots a}]_a
\end{array}
$$

Figure 3.2: ANF transformation for *IDL*

decomposes the term according to the evaluation rules and uses two helper functions. Function $[\cdot]_a$ transforms a continuation expecting an atomic expression (which are created when transforming commands) into one accepting any command by let-binding the passed argument c when necessary. Function $[\![\cdot]\!]_s\, \cdot$ sequences computation of multiple expressions by creating a chain of let-bindings (using $[\cdot]_a$) and then calling the continuation with created variables.

3.2 Control-Flow Analysis

The analysis most relevant to the task of deriving abstract machines from interpreters is the control-flow analysis. Its objective is to find for each expression in a program an over-approximation of a set of functions it may evaluate to [36]. This information can be used in two places: when determining whether a function and applications should be CPS transformed and for checking which functions an expression in operator position may evaluate to. There are a couple of different approaches to performing this analysis available in the literature: abstract interpretation [36], (annotated) type systems [36] and abstract abstract machines [29]. I chose to employ the last approach as it allows for derivation of the control-flow analysis from an abstract machine for

IDL. The derivation technique guarantees correctness of the resulting interpreter and hence provides high confidence in the actual implementation of the machine. I will present the template for acquiring both concrete and abstract versions of the abstract machine for *IDL* but refrain from stepping through the whole derivation. To understand the reasoning and insights behind the technique I highly recommend reading the original work in [29].

A Machine Template

We will begin with a template of a machine for *IDL* terms in A-normal form presented in Figure 3.3. It is a CEK-style machine modified to explicitly allocate memory for values and continuations in an abstract store. The template is parameterized by: implementation of the store σ along with five operations: $alloc_v$, $alloc_k$, $deref_v$, $deref_k$ and $copy_v$; interpretation of primitive operations δ and implementation of *match* function which interprets pattern matching. The store maps value addresses ν to values v and continuation addresses κ to continuations k. The environment maps program variables to value locations. The values on which machine operates are the following: base values b, primitive operations δ, records with addresses as fields, closures and top-level functions. Thanks to terms being in A-normal form, there are only two kinds of continuations which form a stack. The stack frames $\langle \rho, p, e, \kappa \rangle$ are introduced by let-bindings. They hold an environment ρ, a pattern p to use for destructuring of a value, the body e of a let expression and a pointer to the next continuation κ. The bottom of the stack is marked by the empty continuation $\langle\rangle$. We assume that every term has a unique label l which will be used in abstract version of the machine to implement store addresses.

The machine configurations are pairs of a store σ and a partial configuration γ. This split of configuration into two parts will prove beneficial when we will be instantiating the template to obtain an abstract interpreter. There are two classes of partial configurations. An evaluation configuration contains an environment ρ, an expression e and a continuation pointer κ. A continuation configuration holds an address ν of a value that has been computed so far and a pointer κ to a resumption which should be applied next.

The first case of the transition relation $\Rightarrow$ looks up a pointer for the variable x in the environment ρ and switches to the continuation mode. It modifies the store via *copy* function which ensures that every occurrence of a variable has a corresponding binding in the store. The next three cases deal with values by *alloc*ating them in the store and switching to the continuation mode. When the machine encounters a let-binding it allocates a continuation for the body e of the expression and proceeds to evaluate the bound command c with the new pointer κ'. In case of applications and match expressions the resulting configuration is decided using auxiliary functions *apply* and *match*, respectively. Finally, in the continuation mode, the machine may only transition if the continuation loaded from the address κ is a frame. In such

a case the machine matches the stored pattern against the value pointed-to by ν. Otherwise κ points to a $\langle\rangle$ instead and the machine has reached the final state. The auxiliary function *apply* checks what kind of function is referenced by ν and proceeds accordingly.

$$
\begin{array}{rl}
\nu \in \mathit{VAddr} & \kappa \in \mathit{KAddr} \quad l \in \mathit{Label} \quad \sigma \in \mathit{Store} \\
\delta \in \mathit{PrimOp} & \subseteq \mathit{Val}^* \to \mathit{Val} \\
\rho \in \mathit{Env} & = \mathit{Var} \to \mathit{VAddr} \\
\mathit{Val} \ni v & ::= b \mid \delta \mid \{r\ \nu \ldots\} \mid \langle \rho, x \ldots, e \rangle \mid (\texttt{def}\ x\ (x \ldots)\ e) \\
\mathit{Cont} \ni k & ::= \langle \rho, p, e, \kappa \rangle \mid \langle\rangle \\
\mathit{PartialConf} \ni \gamma & ::= \langle \rho, e, \kappa \rangle_e \mid \langle \nu, \kappa \rangle_c \\
\mathit{Conf} \ni \varsigma & ::= \langle \sigma, \gamma \rangle
\end{array}
$$

$$
\begin{array}{rl}
\langle \sigma, \langle \rho, x, \kappa \rangle_e \rangle & \Rightarrow \langle \mathit{copy}_v(\rho(x), l, \sigma), \langle \rho(x), \kappa \rangle_c \rangle \\
\langle \sigma, \langle \rho, b^l, \kappa \rangle_e \rangle & \Rightarrow \langle \sigma', \langle \nu, \kappa \rangle_c \rangle \\
 & \text{where } \langle \sigma', \nu \rangle = \mathit{alloc}_v(b, l, \sigma) \\
\langle \sigma, \langle \rho, \{r\ x \ldots\}^l, \kappa \rangle_e \rangle & \Rightarrow \langle \sigma', \langle \nu, \kappa \rangle_c \rangle \\
 & \text{where } \langle \sigma', \nu \rangle = \mathit{alloc}_v(\{r\ \rho(x) \ldots\}, l, \sigma) \\
\langle \sigma, \langle \rho, (\texttt{fun}\ (x \ldots) e)^l, \kappa \rangle_e \rangle & \Rightarrow \langle \sigma', \langle \nu, \kappa \rangle_c \rangle \\
 & \text{where } \langle \sigma', \nu \rangle = \mathit{alloc}_v(\langle \rho, x \ldots, e \rangle, l, \sigma) \\
\langle \sigma, \langle \rho, (\texttt{let}\ p\ c^l\ e), \kappa \rangle_e \rangle & \Rightarrow \langle \sigma', \langle \rho, c, \kappa' \rangle_e \rangle \\
 & \text{where } \langle \sigma', \kappa' \rangle = \mathit{alloc}_k(\langle \rho, p, e, \kappa \rangle, l, \sigma) \\
\langle \sigma, \langle \rho, (x\ y \ldots), \kappa \rangle_e \rangle & \Rightarrow \mathit{apply}(\sigma, \rho(x), \rho(y) \ldots, l) \\
\langle \sigma, \langle \rho, (\texttt{match}\ x\ (p\ e) \ldots), \kappa \rangle_e \rangle & \Rightarrow \mathit{match}(\sigma, \rho, \rho(x), \langle p, e \rangle \ldots) \\
\langle \sigma, \langle \nu, \kappa \rangle_c \rangle & \Rightarrow \mathit{match}(\sigma, \rho, \nu, \kappa', \langle p, e \rangle) \\
 & \text{where } \langle \rho, p, e, \kappa' \rangle = \mathit{deref}_k(\sigma, \kappa)
\end{array}
$$

$$
\mathit{apply}(\sigma, \nu, \nu' \ldots, \kappa, l) = \begin{cases}
\langle \sigma, \langle \rho[(x \mapsto \nu') \ldots], e, \kappa \rangle_e \rangle \\
\quad \text{when } \mathit{deref}_v(\sigma, \nu) = \langle \rho, x \ldots, e \rangle \\
\langle \sigma, \langle \rho_0[(x \mapsto \nu') \ldots], e, \kappa \rangle_e \rangle \\
\quad \text{when } \mathit{deref}_v(\sigma, \nu) = (\texttt{def}\ y\ (x \ldots)\ e) \\
\langle \sigma', \langle \nu'', \kappa \rangle_c \rangle \\
\quad \text{when } \mathit{deref}_v(\sigma, \nu) = \delta \\
\quad \text{and } \langle \sigma', \nu'' \rangle = \mathit{alloc}_v(\delta(\sigma(\nu') \ldots), l, \sigma)
\end{cases}
$$

$$
\mathit{match}(\sigma, \rho, \nu, \kappa, \langle p, e \rangle \ldots) = \langle \sigma, \langle \rho', e', \kappa \rangle_e \rangle \text{ where } \rho' \text{ is the environment for the first matching branch with body } e'
$$

Figure 3.3: An abstract machine for *IDL* terms in ANF

A Concrete Abstract Machine

The machine template can now be instantiated with a store, a *match* implementation which finds the first matching branch and interpretation for primitive operations in

order to obtain an abstract machine. By choosing *Store* to be a mapping with infinite domain we can ensure that *alloc* can always return a fresh address. In this setting the store-allocated continuations are just an implementation of a stack. The extra layer of indirection introduced by storing values in a store can also be disregarded as the machine operates on persistent values. Therefore the machine corresponds to a CEK-style abstract machine which is a natural [19] formulation for call-by-value functional calculi.

An Abstract Abstract Machine

Let us now turn to a different instantiation of the template. Figure 3.4 shows the missing pieces of an abstract abstract machine for *IDL*. The abstract values use base type names *tp* to represent any value of that type, abstract versions of primitive operations, records, closures and top-level functions. The interpretation of primitive operations must approximate their concrete counterparts.

The store is represented as a pair of finite mappings from labels to sets of abstract values and continuations, respectively. This bounding of store domain and range ensures that the state-space of the machine becomes finite and therefore can be used for computing an analysis. To retain soundness w.r.t. the concrete abstract machine the store must map a single address to multiple values to account for address reuse. This style of abstraction is fairly straightforward as noted by [29] and used in textbooks [36]. When instantiated with this store, the transition relation $\Rightarrow$ becomes nondeterministic as pointer *deref*erencing nondeterministically returns one of the values available in the store. Additionally the implementation of *match* function is also nondeterministic in choice of a branch to match against. This machine is not yet suitable for computing the analysis as the state space is still too large since every machine configuration has its own copy of the store. To circumvent this problem a standard technique of widening [36] can be employed. In particular, following [29], we will use a global store. The abstract configuration $\tilde{\varsigma}$ is a pair of a store and a set of partial configurations. The abstract transition $\Rightarrow_a$ performs one step of computation using $\Rightarrow$ on the global store σ paired with every partial configuration γ. The resulting stores σ' are merged together and with the original store to create a new, extended global store. The partial configurations C' are added to the initial set of configurations C. The transition relation $\Rightarrow_a$ is deterministic so it can be treated as a function. This function is monotone on a finite lattice and therefore is amenable to fixed-point iteration.

Computing the Analysis

With the abstract transition function in hand we can now specify the algorithm for obtaining the analysis. To start the abstract interpreter we must provide it with an initial configuration: a store, an environment, a term and a continuation pointer. The

$$
\begin{array}{rl}
\mathit{VAddr} & = \mathit{KAddr} = \mathit{Label} \\
\widetilde{\mathit{Val}} \ni v & ::= tp \mid \widetilde{\delta} \mid \{r\ \nu \ldots\} \mid \langle \rho, x \ldots, e \rangle \mid (\texttt{def}\ x\ (x \ldots)\ e) \\
\sigma \in \mathit{Store} & = (\mathit{VAddr} \to \mathbb{P}(\widetilde{\mathit{Val}})) \times (\mathit{KAddr} \to \mathbb{P}(\mathit{Cont})) \\
\mathit{alloc}_v(v, l, \langle \sigma_v, \sigma_k \rangle) & = \langle \langle \sigma_v[l \mapsto \sigma_v(l) \cup \{v\}], \sigma_k \rangle, l \rangle \\
\mathit{alloc}_k(v, l, \langle \sigma_v, \sigma_k \rangle) & = \langle \langle \sigma_v, \sigma_k[l \mapsto \sigma_k(l) \cup \{k\}] \rangle, l \rangle \\
\mathit{copy}_v(\nu, l, \langle \sigma_v, \sigma_k \rangle) & = \langle \sigma_v[l \mapsto \sigma_v(l) \cup \sigma_v(\nu)], \sigma_k \rangle \\
\mathit{deref}_v(l, \langle \sigma_v, \sigma_k \rangle) & = \sigma_v \\
\tilde{\varsigma} \in \widetilde{\mathit{Conf}} & = \mathit{Store} \times \mathbb{P}(\mathit{PartialConf}) \\
\hline
\langle \sigma, C \rangle & \Rightarrow_a \langle \sigma' \sqcup \sigma, C \cup C' \rangle \\
 & \text{where } \sigma' = \bigsqcup \{\sigma' \mid \exists \gamma \in C.\, \langle \sigma, \gamma \rangle \Rightarrow \langle \sigma', \gamma' \rangle\} \\
 & \text{and } C' = \{\gamma' \mid \exists \gamma \in C.\, \langle \sigma, \gamma \rangle \Rightarrow \langle \sigma', \gamma' \rangle\}
\end{array}
$$

Figure 3.4: An abstract abstract machine for *IDL*

store will be assembled from datatype and structure definitions of the program as well as base types. The initial term is the body of the `main` function of the interpreter and the environment is the global environment extended with `main`'s parameters bound to pointers to datatypes in the above-built store. The initial continuation is of course $\langle\rangle$ and the pointer is the label of the `main`'s body. The analysis is computed by performing fixed-point iteration of $\Rightarrow_a$. The resulting store will contain a set of functions to which every variable (the only allowed term) in function position may evaluate (ensured by the use of copy_v function). This result will be used in Sections 3.3 and 3.4.

3.3 Selective CPS

In this section we will formulate an algorithm for selectively transforming the program into continuation-passing style. All functions (both anonymous and top-level) marked `#:atomic` by the user will be kept in direct style. The `main` function is implicitly marked as atomic since its interface should be preserved as it is an entry point of the interpreter. Primitive operations are treated as atomic at call-site. Atomic functions may call non-atomic ones by providing the called function an identity continuation. The algorithm uses the results of control-flow analysis to determine whether all functions to which a variable labeled l in function position may evaluate are atomic – denoted $\mathit{allAtomic}(l)$ or none of them are atomic – $\mathit{noneAtomic}(l)$. When both atomic and non-atomic functions may be called the algorithm cannot proceed and signals an error in the source program.

The algorithm consists of two mutually recursive transformations: $[\![e]\!]_c\, k$ in Figure 3.5 transforming a term e into CPS with a program variable k as a continuation and $[\![e]\!]_d$ in Figure 3.6 transforming a term e which should be kept in direct style.

The first five clauses of the CPS translation deal with values. When a variable

$$
\begin{aligned}
[\![x]\!]_c\, k &= (k\ x) \\
[\![b]\!]_c\, k &= (\texttt{let}\ x\ b\ (k\ x)) \\
[\![\{r\ x\ldots\}]\!]_c\, k &= (\texttt{let}\ y\ \{r\ x\ldots\}\ (k\ y)) \\
[\![(\texttt{fun}\ \texttt{\#:atomic}\ (x\ldots) e)]\!]_c\, k &= (\texttt{let}\ y\ (\texttt{fun}\ (x\ldots)\ [\![e]\!]_d)\ (k\ y)) \\
[\![(\texttt{fun}\ (x\ldots) e)]\!]_c\, k &= (\texttt{let}\ y\ (\texttt{fun}\ (x\ldots k')\ [\![e]\!]_c\, k')\ (k\ y)) \\
[\![(f^l\ x\ldots)]\!]_c\, k &= \begin{cases} (f\ x\ldots\ k) & \text{when } \mathit{noneAtomic}(l) \\ (\texttt{let}\ y\ (f\ x\ldots)\ (k\ y)) & \text{when } \mathit{allAtomic}(l) \end{cases} \\
[\![(\texttt{match}\ x\ (p\ e)\ldots)]\!]_c\, k &= (\texttt{match}\ x\ (p\ [\![e]\!]_c\, k)\ldots) \\
[\![(\texttt{let}\ x\ c\ e)]\!]_c\, k &= \begin{cases} (\texttt{let}\ x\ [\![d]\!]_d\ [\![e]\!]_c\, k) & \text{when } \mathit{trivial}(c) \\ (\texttt{let}\ k'\ (\texttt{fun}\ (x)\ [\![e]\!]_c\, k)\ [\![c]\!]_c\, k') & \text{otherwise} \end{cases} \\
[\![(\texttt{error}\ s)]\!]_c\, k &= (\texttt{error}\ s)
\end{aligned}
$$

Figure 3.5: A translation for CPS terms

$$
\begin{aligned}
[\![x]\!]_d &= x \\
[\![b]\!]_d &= b \\
[\![\{r\ x\ldots\}]\!]_d &= \{r\ x\ldots\} \\
[\![(\texttt{fun}\ \texttt{\#:atomic}\ (x\ldots) e)]\!]_d &= (\texttt{fun}\ (x\ldots)\ [\![e]\!]_d) \\
[\![(\texttt{fun}\ (x\ldots) e)]\!]_d &= (\texttt{fun}\ (x\ldots k')\ [\![e]\!]_c\, k') \\
[\![(f^l\ x\ldots)]\!]_d &= \begin{cases} (f\ x\ldots) & \text{when } \mathit{allAtomic}(l) \\ (\texttt{let}\ k\ (\texttt{fun}\ (y)\ y)\ (f\ x\ldots\ k)) & \text{when } \mathit{noneAtomic}(l) \end{cases} \\
[\![(\texttt{match}\ x\ (p\ e)\ldots)]\!]_d &= (\texttt{match}\ x\ (p\ [\![e]\!]_d)\ldots) \\
[\![(\texttt{let}\ x\ (f^l\ y\ldots)\ e)]\!]_d &= \begin{array}{l} (\texttt{let}\ k\ (\texttt{fun}\ (z)\ z) \\ \quad (\texttt{let}\ x\ (f\ y\ldots\ k)) \\ \qquad [\![e]\!]_d) \end{array} \quad \text{when } \mathit{noneAtomic}(l) \\
[\![(\texttt{let}\ x\ c\ e)]\!]_d &= (\texttt{let}\ x\ [\![c]\!]_d\ [\![e]\!]_d) \\
[\![(\texttt{error}\ s)]\!]_d &= (\texttt{error}\ s)
\end{aligned}
$$

Figure 3.6: A translation for terms which should be left in direct style

is encountered it may be immediately returned by applying a continuation. In other cases the value must be let-bound in order to preserve the A-normal form of the term and then the continuation is applied to the introduced variable. The body e of an anonymous function is translated using $[\![e]\!]_d$ when the function is marked atomic. When the function is not atomic a new variable k' is appended to its parameter list and its body is translated using $[\![e]\!]_c\, k'$. The form of an application depends on atomicity of functions which may be applied. When none of them are atomic the continuation k is passed to the function. When all of them are atomic the result of the call is let-bound and returned by applying the continuation k. Match expression is transformed by recursing on its branches. Since the continuation is always a program variable no code gets duplicated. When transforming a let expression the algorithm checks whether the bound command c is *trivial* – meaning it will only

$$
\begin{aligned}
\mathit{trivial}(x) \quad & \mathit{trivial}(b) \\
\mathit{trivial}(\texttt{(\{}r\ x \ldots\texttt{\})}) \quad & \mathit{trivial}(\texttt{(fun (}x \ldots\texttt{)}e\texttt{)}) \\
\mathit{trivial}(\texttt{(}f^l\ x \ldots\texttt{)}) & \iff \mathit{allAtomic}(l) \\
\mathit{trivial}(\texttt{(match }x\texttt{ (}b\ e\texttt{)}\ldots\texttt{)}) & \iff \bigwedge \mathit{trivial}(e) \ldots \\
\mathit{trivial}(\texttt{(let }x\ c\ e\texttt{)}) & \iff \mathit{trivial}(c) \wedge \mathit{trivial}(e)
\end{aligned}
$$

Figure 3.7: The *trivial* predicate

call atomic functions when evaluated (defined in Figure 3.7). If it is then it can remain in direct style $[\![c]\!]_d$, no new continuation has to be introduced and the body can be transformed by $[\![e]\!]_c\, k$. If the command is non-trivial then a new continuation is created and bound to k'. This continuation uses the variable x as its argument and its body is the body of let-expression e transformed with the input continuation k. The bound command is transformed with the newly introduced continuation k'. Finally, the translation of `error` throws out the continuation.

The transformation for terms which should be kept in direct style begins similarly to the CPS one – with five clauses for values. In case of an application the algorithm considers two possibilities: when all functions are atomic the call remains in direct style, when none of them are atomic a new identity continuation k is constructed and is passed to the called function. A match expression is again transformed recursively. A let binding of a call to cps function gets special treatment to preserve A-normal by chaining allocation of identity continuation with the call. In other cases a let binding is transformed recursively. An `error` expression is left untouched.

Each top-level function definition in a program is transformed in the same fashion as anonymous functions. After the transformation the program is still in ANF and can be again analyzed by the abstract abstract machine of the previous section.

3.4 Selective Defunctionalization

The second step of the functional correspondence and the last stage of the transformation is selective defunctionalization. The goal is to defunctionalize function spaces deemed interesting by the author of the program. To this end top-level and anonymous functions may be annotated with `#:no-defun` to skip defunctionalization of function spaces they belong to. In the algorithm of Figure 3.8 the predicate *defun* specifies whether a function should be transformed. Predicates *primOp* and *topLevel* specify whether a variable refers to (taking into account the scoping rules) primitive operation or top-level function, respectively. For each application point l in the program we can utilize the results of control-flow analysis to obtain the set of

$$
\begin{aligned}
[\![x]\!] &= \begin{cases} \{\texttt{Prim}_x\} & \text{when } primOp(x) \\ \{\texttt{Top}_x\} & \text{when } topLevel(x) \\ x & \text{otherwise} \end{cases} \\
[\![b]\!] &= b \\
[\![\{r\ x \ldots\}]\!] &= \{\texttt{r}\ [\![x]\!] \ldots\} \\
[\![(\texttt{fun}\ (x \ldots) e)^l]\!] &= \begin{cases} \{\texttt{Fun}_l\ fvs(e)\} & \text{when } defun(l) \\ (\texttt{fun}\ (x \ldots)\ [\![e]\!]) & \text{otherwise} \end{cases} \\
[\![(f^{l'}\ x \ldots)^l]\!] &= \begin{cases} (\texttt{apply}_l\ f\ [\![x]\!] \ldots) & \text{when } allDefun(l') \\ (f\ [\![x]\!] \ldots) & \text{when } noneDefun(l') \end{cases} \\
[\![(\texttt{match}\ x\ (p\ e) \ldots)]\!] &= (\texttt{match}\ x\ (p\ [\![e]\!]) \ldots) \\
[\![(\texttt{let}\ x\ c\ e)]\!] &= (\texttt{let}\ x\ [\![c]\!]\ [\![e]\!]) \\
[\![(\texttt{error}\ s)]\!] &= (\texttt{error}\ s)
\end{aligned}
$$

Figure 3.8: Selective defunctionalization algorithm for *IDL*

$$
\begin{aligned}
mkBranch(x \ldots, \qquad\qquad \delta) &= (\{\texttt{Prim}_\delta\}\ (\delta\ x \ldots)) \\
mkBranch(x \ldots, (\texttt{def}\ f\ (y \ldots)\ \texttt{e})) &= (\{\texttt{Top}_f\}\ (f\ x \ldots)) \\
mkBranch(x \ldots, \quad (\texttt{fun}\ (y \ldots)\ e)^l) &= (\{\texttt{Fun}_l\ fvs(e)\}\ [\![e]\!]\,[y \mapsto x])
\end{aligned}
$$

$$
\begin{aligned}
mkApply(l, fn \ldots) = \quad & (\texttt{def}\ \texttt{apply}_l\ (f\ x \ldots) \\
& \quad (\texttt{match}\ f \\
& \qquad mkBranch(x \ldots, fn) \ldots))
\end{aligned}
$$

Figure 3.9: Top-level apply function generation

functions which may be applied. If all of them should be defunctionalized (*allDefun*) then a call to the generated apply function is introduced, when none of them should (*noneDefun*) then the application is left as is, if neither condition holds then an error in the source program is signaled. The apply functions are generated using *mkApply* as specified in Figure 3.9 where the *fn* ... is a list of functions which may be applied. After the transformation the program is no longer in A-normal form since variables referencing top-level functions may have been transformed into records. However it does not pose a problem since the majority of work has already been done and the last step – let-inlining does not require the program to be in ANF.

Chapter 4

Case Studies

I studied the performance of the algorithm and the implementation on a number of programming language calculi. Figure 4.1 shows a summary of interpreters on which I tested the transformer. The first group of interpreters is denotational (mostly meta-circular) in style and covers various extensions of the base λ-calculus with call-by-value evaluation order. The additions I tested include: integers with addition, recursive let-bindings, delimited control operators – *shift* and *reset* with CPS interpreter based on [25] and exceptions in two styles: monadic with exceptions as values (functions return either value or an exception) and in CPS with success and error continuations. The last interpreter for call-by-value in Figure 4.1 is a normalization function based on normalization by evaluation technique transcribed from [37]. The next three interpreters correspond to big-step operational semantics for call-by-name λ-calculus, call-by-need (call-by-name with memoization) and a simple imperative language, respectively.

Transformation of call-by-value and call-by-need λ-calculus yielded machines very similar to the CEK and Krivine machines, respectively. I was also able to replicate the machines previously obtained via manual application of the functional correspondence ([19, 25, 24]). The biggest differences were due to introduction of administrative transitions in handling of applications. This property hints at a potential for improvement by introducing an inlining step to the transformation. An interesting feature of the transformation is the ability to select which parts of the interpreter should be transformed and which should be considered atomic. These choices are reflected in the resulting machine, e.g., by transforming an environment look up in call-by-need interpreter we obtain a Krivine machine which has the search for a value in the environment embedded in its transition rules, while marking it atomic gives us a more abstract formulation from [19]. Another consequence of this feature is that one can work with interpreters already in CPS and essentially skip directly to defunctionalization (as tested on micro-Prolog interpreter of [24]).

In the remainder of this chapter I will present three case studies. The first describes an interpreter encoding the natural semantics of a simple imperative language

Language	Interpreter style	Lang. Features	Result
call-by-value λ-calculus	denotational	·	CEK machine
	denotational	integers with add	CEK with add
	denotational, recursion via environment	integers, recursive let-bindings	similar to Reynold's first-order interpreter
	denotational with conts.	shift and reset	two layers of conts.
	denotational, monadic	exceptions with handlers	explicit stack unwinding
	denotational, CPS		pointer to exception handler
	normalization by evaluation	·	strong CEK machine
call-by-name λ-calculus	big-step	·	Krivine machine
call-by-need λ-calculus	big-step (state passing)	memoization	lazy Krivine machine
simple imperative	big-step (state passing)	conditionals, while, assignment	·
micro-Prolog	CPS	backtracking, cut operator	logic engine

Figure 4.1: Summary of tested interpreters

and serves as a demonstration of general properties of the transformation and machines it produces. The second example is of an interpreter for λ-calculus extended with exceptions and exception handlers which shows how additions to the source semantics translate to changes in the resulting machine. The last example shows that the transformation can be applied to semantic specifications stronger than evaluators. To this end I show a derivation of a machine performing applicative order normalization of λ-terms from a high-level normalization function.

4.1 Natural semantics for an imperative language

Let us begin with an encoding of natural semantics for a simple imperative language shown in Figure 4.2. The first two functions provide an implementation of memory (a mapping from locations to values – integers) as a function. Since we are not interested in implementation strategy of the memory, the function `init-state` and the anonymous function beginning in 4th line are both annotated with `#:atomic` and `#:no-defun`. Together with `#:atomic` annotation on `update-state` it ensures that

```
(def init-state #:atomic #:no-defun (var) 0)

(def update-state #:atomic (tgt val state)
  (fun #:atomic #:no-defun (var)
    (match (eq? tgt var)
      (#t val)
      (#f (state var)))))

(def aval #:atomic (state aexpr)
  (match aexpr
    ([Integer n] n)
    ([String var] (state var))
    ({Add aexpr1 aexpr2}
      (+ (aval state aexpr1) (aval state aexpr2)))))

(def bval #:atomic (state bexpr)
  (match bexpr
    ({Eq aexpr1 aexpr2}
      (eq? (aval state aexpr1) (aval state aexpr2)))))

(def eval (state cmd)
  (match cmd
    ({Skip} state)
    ({Assign var aexpr}
      (update-state var (aval state aexpr) state))
    ({If cond then else}
      (match (bval state cond)
        (#t (eval state then))
        (#f (eval state else))))
    ({Seq cmd1 cmd2}
      (let state (eval state cmd1))
      (eval state cmd2))
    ({While cond cmd}
      (match (bval state cond)
        (#t
          (let state (eval state cmd))
          (eval state {While cond cmd}))
        (#f state)))))

(def main ([Cmd cmd])
  (eval init-state cmd))
```

Figure 4.2: An encoding of natural semantics for an imperative language

```
1  (def eval (state cmd cont)
2    (match cmd
3      ({Skip } (continue cont state))
4      ({Assign var aexpr}
5        (continue cont (update-state var (aval state aexpr) state)))
6      ({If cond then else}
7        (match (bval state cond)
8          (#t (eval state then cont))
9          (#f (eval state else cont))))
10     ({Seq cmd1 cmd2} (eval state cmd1 {Seq1 cmd2 cont}))
11     ({While cond cmd}
12       (match (bval state cond)
13         (#t (eval state cmd {While1 cmd cond cont}))
14         (#f (continue cont state))))))
15
16 (def continue (fn state)
17   (match fn
18     ({Seq1 cmd2 cont} (eval state cmd2 cont))
19     ({While1 cmd cond cont} (eval state {While cond cmd} cont))
20     ({Halt } state)))
21
22 (def main ([Cmd cmd]) (eval init-state cmd {Halt }))
```

Figure 4.3: An abstract machine for the simple imperative language

after the transformation, operations on memory (both updates and lookups) will be treated as builtin operations of the machine. Following the natural semantics, functions `aval` and `bval` define valuations of arithmetic and boolean expressions. They are also marked as `#:atomic` since they are pure and side-effect free. The `eval` function is a straightforward translation of the deduction rules of the natural semantics. It retains the state passing style and usage of semantic functions for valuation of expressions. The rules are encoded as branches of pattern matching on syntax of a command. The branch for assignment updates the state with valuation of an arithmetic expression. The rules for `If` command are both encoded as a pattern match on truthiness of valuation of the condition. The branch for `Seq` command sequences the computation by passing state updated by evaluating `cmd1` to evaluation of `cmd2`. The branch for `While` loop encodes the two rules of the big-step semantics by checking whether the condition holds and either executing the body of the loop and recursively evaluating the whole loop in a new state or returning the state.

The machine we obtained is presented in Figure 4.3. The figure consists of three functions: `eval` and `main` which were changed during the transformation and a generated `continue` function. The rest of the interpreter remains unchanged (modulo

omitted record definitions). We can see that the machine uses a stack of continuations to handle sequencing of commands (`Seq1`) and iteration (`While1`) in while loops. Even though loops introduce a stack frame, only one frame is present on the stack for a given `While` command regardless of number of iterations. This frame is responsible for continuing the iteration after the body of the loop has been executed. The final continuation `Halt` is provided by the `main` function which builds the initial configuration and starts the machine. The clause for `Halt` extracts the result from the final configuration.

The names for continuations, the `continue` function as well as its parameters have been generated without any assistance from the user and contribute to the readability of the machine. The machine is formulated in a natural way, considering it works directly on abstract syntax of the language rather than a series of instructions. This manifests in usage of runtime continuation stack instead of *goto* instructions directly in the instruction sequence.

4.2 Call-by-value λ-calculus with exceptions

The second example is an interpreter for the λ-calculus extended with exceptions which may be thrown and (optionally) caught and with simple arithmetic. The interpreter shown in Figure 4.4 can be considered an extension of the meta-circular interpreter from Chapter 1. The functions are now embedded in the standard "either" monad: they return either {`Ok value`} or an {`Err value`}. The sequencing of computation requires pattern matching on the result and either propagating an error or continuing with the value. It is apparent in branches for `App` and `Add`. The values of this interpreter – integers and functions, are returned in the `Ok` variant to indicate success; same holds for values looked up in the environment. The `Raise` expression introduces errors by returning the number `n` in the `Err` variant. This limitation to constants is imposed in order to keep the listing one page long. In the branch for `Try`, first an expression `t` is executed and in case of exception the handler `handle` will be evaluated in an environment extended with the value passed by the exception. We are not interested in control-flow behavior of environment extension or look up so they are all marked `#:atomic`. We allow them to be defunctionalized and specify that the resulting record for anonymous function in line 29 should be called `Extend` and the apply function should be named `lookup`.

The transformed interpreter is presented in Figure 4.5. We can see that defunctionalized environment forms a list of variable-value pairs. Instead of environment application the `lookup` function is called which recursively searches for the binding. The function is kept in direct style as intended.

Let us now turn to the transition functions: `eval` and `continue`. The sequencing of operations has been split between the two functions. To evaluate an application in the object-language first the `fn` expression is evaluated (line 6) with `App1` continuation

```
(def eval (env [Term term])
  (match term
    ([String x] {Ok (env x)})
    ([Integer n] {Ok n})
    ({Lam x body}
     {Ok (fun (v) (eval (extend env x v) body))})
    ({App fn arg}
      (match (eval env fn)
        ({Err e} {Err e})
        ({Ok f}
          (match (eval env arg)
            ({Err e} {Err e})
            ({Ok v} (f v))))))
    ({Add n m}
      (match (eval env n)
        ({Err e} {Err e})
        ({Ok n}
          (match (eval env m)
            ({Err e} {Err e})
            ({Ok m} {Ok (+ n m)})))))
    ({Raise n} {Err n})
    ({Try t x handle}
      (match (eval env t)
        ({Ok v} {Ok v})
        ({Err e} (eval (extend env x e) handle))))
    ))

(def extend #:atomic (env k v)
  (fun #:atomic #:name Extend #:apply lookup (x)
    (match (eq? x k)
      (#t v)
      (#f (env x)))))

(def init #:atomic (x) (error "empty environment"))

(def main ([Term term])
  (eval init term))
```

Figure 4.4: An interpreter for λ-calculus with exceptions and arithmetic

```
(def eval (env [Term term] cont)
  (match term
    ([String x] (continue1 cont {Ok (lookup env x)}))
    ([Integer n] (continue1 cont {Ok n}))
    ({Lam x body} (continue1 cont {Ok {Fun body env x}}))
    ({App fn arg} (eval env fn {App1 arg cont env}))
    ({Add n m} (eval env n {Add1 cont env m}))
    ({Raise n} (continue1 cont {Err n}))
    ({Try t x handle} (eval env t {Try1 cont env handle x}))))

(def extend (env k v) {Extend env k v})

(def init (x) (error "empty environment"))

(def apply (fn1 v cont1)
  (match fn1 ({Fun body env x} (eval (extend env x v) body cont1))))

(def continue1 (fn2 var3)
  (match fn2
    ({Ok1 cont f}
      (match var3
        ({Err e} (continue1 cont {Err e}))
        ({Ok v} (apply f v cont))))
    ({App1 arg cont env}
      (match var3
        ({Err e} (continue1 cont {Err e}))
        ({Ok f} (eval env arg {Ok1 cont f}))))
    ({Ok2 cont n}
      (match var3
        ({Err e} (continue1 cont {Err e}))
        ({Ok m} (continue1 cont {Ok (+ n m)}))))
    ({Add1 cont env m}
      (match var3
        ({Err e} (continue1 cont {Err e}))
        ({Ok n} (eval env m {Ok2 cont n}))))
    ({Try1 cont env handle x}
      (match var3
        ({Ok v} (continue1 cont {Ok v}))
        ({Err e} (eval (extend env x e) handle cont))))
    ({Halt } var3)))

(def lookup (fn3 x)
  (match fn3
    ({Extend env k v}
      (match (eq? x k)
        (#t v)
        (#f (lookup env x))))
    ({Init } (init x))))

(def main ([Term term]) (eval {Init } term {Halt }))
```

Figure 4.5: A machine for λ-calculus with exceptions and arithmetic

pushed onto the stack. When the control reaches this frame (lines 24-27), `var3` contains either an exception or the computed value for the function. In the `Ok` case, the argument `arg` and the environment `env` are restored and evaluated with `Ok1` frame pushed onto the stack. Finally, when control reaches `Ok1` and `var3` holds a value, the function value is popped from the stack and applied to the argument (line 23). Looking at the `continue` function we can see that if it is passed an exception (`Err` variant) then it will unroll the continuation stack until it finds either a `Try1` frame or the bottom of the stack – `Halt`. In case of the latter the machine finishes execution with `Err` value. In case of the former a handling expression is evaluated in the saved environment extended with the exception code.

4.3 Normalization by Evaluation for λ-calculus

The last case study is of a normalization function rather than a typical interpreter. Figure 4.6 contains the development. The technique used is called normalization by evaluation and the particular definition has been adapted from [37]. The main approach is to use standard evaluator for call-by-value λ-calculus to evaluate terms to values and then reify them back into terms. The terms use de Bruijn indices to represent bound variables. Since normalization requires reduction under binders the evaluator must work with open terms. We will use de Bruijn levels (`Level`) to model variables in open terms. The `eval` function as usual transforms a term in a given environment into a value which is represented as a function wrapped in a `Fun` record. The values also include `Level`s and `Term`s which are introduced by the `reify` function. The `apply` function handles both the standard case of applying a functional value (case `Fun`) and the non-standard one which occurs during reification of the value and amounts to emitting the syntax node for application. The reification function (`reify`) turns a value back into a term. When its argument is a `Fun` it applies the function `f` to a `Level` representing unknown variable. When reified, a `Level` is turned back into de Bruijn index. Lastly, reification of an (syntactic) application proceeds recursively. The `main` function first evaluates a term in an empty environment and then reifies it back into its normal form. As usual, we will keep the environment implementation unchanged during the transformation and we annotate the functional values to be named `Closure`.

The transformed normalization function is presented in Figure 4.7. We notice that the machine has two classes of continuations. The first set (handled by `continue1`) is responsible for the control-flow of reification procedure. The second set (handled by `continue`) is responsible for the control-flow of evaluation and for switching to reification mode. We observe that the stack used by the machine consists of a prefix of only evaluation frames and a suffix of only reification frames. The machine switches between evaluation and reification in three places. In line 3 reification of a closure requires evaluation of its body therefore machine uses `apply1` to evaluate the closure with a `Level` as an argument. The switch in other direction

```
1  (def-data Term
2    {Var Integer}
3    {App Term Term}
4    {Abs Term})
5
6  (def-struct {Level Integer})
7  (def-struct {Fun Any})
8
9  (def cons #:atomic (val env)
10   (fun #:atomic #:no-defun (n)
11     (match n
12       (0 val)
13       (_ (env (- n 1))))))
14
15 (def reify (ceil val)
16   (match val
17     ({Fun f}
18       {Abs (reify (+ ceil 1) (f {Level ceil}))})
19     ({Level k} {Var (- ceil (+ k 1))})
20     ({App f arg} {App (reify ceil f) (reify ceil arg)})))
21
22 (def apply (f arg)
23   (match f
24     ({Fun f} (f arg))
25     (_ {App f arg})))
26
27 (def eval (expr env)
28   (match expr
29     ({Var n} (env n))
30     ({App f arg} (apply (eval f env) (eval arg env)))
31     ({Abs body} {Fun (fun #:name Closure (x) (eval body (cons x env)))})))
32
33 (def run (term)
34   (reify 0
35     (eval term (fun #:atomic #:no-defun (x) (error "empty env")))))
36
37 (def main ([Term term]) (run term))
```

Figure 4.6: A normalization function for call-by-value λ-calculus

```
(def reify (ceil val cont)
  (match val
    ({Fun f} (apply1 f {Level ceil} {Fun1 cont (+ ceil 1)}))
    ({Level k} (continue1 cont {Var (- ceil (+ k 1))}))
    ({App f arg} (reify ceil f {App1 arg ceil cont}))))

(def apply (f arg cont1)
  (match f
    ({Fun f} (apply1 f arg cont1))
    (_ (continue cont1 {App f arg}))))

(def eval (expr env cont2)
  (match expr
    ({Var n} (continue cont2 (env n)))
    ({App f arg} (eval f env {App3 arg cont2 env}))
    ({Abs body} (continue cont2 {Fun {Closure body env}}))))

(def run (term cont4) (eval term (fun (x) (error "empty env")) {Cont cont4 0}))

(def apply1 (fn x cont3)
  (match fn ({Closure body env} (eval body (cons x env) cont3))))

(def continue1 (fn1 var11)
  (match fn1
    ({Fun2 cont} (continue1 cont {Abs var11}))
    ({App2 cont var10} (continue1 cont {App var10 var11}))
    ({App1 arg ceil cont} (reify ceil arg {App2 cont var11}))
    ({Halt } var11)))

(def continue (fn2 var13)
  (match fn2
    ({Fun1 cont var3} (reify var3 var13 {Fun2 cont}))
    ({App4 cont2 var12} (apply var12 var13 cont2))
    ({App3 arg cont2 env} (eval arg env {App4 cont2 var13}))
    ({Cont cont4 var16} (reify var16 var13 cont4))))

(def main ([Term term]) (run term {Halt }))
```

Figure 4.7: A strong call-by-value machine for λ-calculus

is due to evaluation finishing: in line 32 a closure's body has been evaluated and has to be reified and then enclosed in an `Abs` (enforced by the `Fun2` frame); in line 35 the initial term has been reduced and the value can be reified. The machine we obtained has, to my knowledge, not been described in the literature. It is somewhat similar to the one obtained by Ager et al. [21] who also used the functional correspondence to derive the machine. Their machine uses meta-language with mutable state in order to generate fresh identifiers for variables in open terms instead of de Bruijn levels and it operates on compiled rather than source terms. The machine we obtained using `semt` can also be considered more legible due to naming of both source and result configuration and marginally better names of continuations.

Chapter 5

Conclusions

In this thesis I described an algorithm allowing for automatic derivation of an abstract machine given an interpreter which usually corresponds to denotational or natural semantics. The algorithm allows for fine-grained control over the shape of the resulting machine. One may annotate functions which should be considered atomic (i.e., they do not contribute to the control-flow of the machine) and function spaces which should be left abstract (i.e, not defunctionalized). In order to enable the transformation I derived the control-flow analysis for *IDL* using the abstracting abstract machines methodology. I implemented the algorithm in the *Haskell* programming language and used this tool to transform a selection of interpreters.

The correctness of the tool has been established experimentally by running the interpreters after every intermediate step of transformation. The next logical step is a formalization of the algorithm using a proof assistant (e.g., Coq) to obtain a powerful and correct method of deriving abstract machines.

In order to extend capabilities of `semt` as a practical tool for semantics engineering the future work could include extending the set of primitive operations and adding the ability to import arbitrary *Racket* functions and provide their abstract specification. Another important matter is the performance (i.e., speed) of the tool. To this end a thorough investigation of cost and complexity of computing the control-flow analysis is required. The tool could also be extended to accommodate other output formats such as LaTeX figures or low level *C* code.

Other avenue for improvement lies in extensions of the meta-language capabilities. Investigation of additions such as control operators, nondeterministic choice and concurrency could yield many opportunities for diversifying the set of interpreters (and languages) that may be encoded in the *IDL*. In particular control operators could allow for expressing the interpreter for a language with delimited control (or algebraic effects) in direct style.

Appendix A

User's Manual

Installation

The semantic transformer – `semt` is built from source using `cabal` – a *Haskell* package manager. To build the binary use `cabal build` and to install the resulting binary use `cabal install` in the root of the project. Use `make test` to test the tool on the provided interpreters which are located in `interpreters/src` directory.

Tool usage

The basic mode of usage is to transform a `file.rkt` containing an interpreter into `out/file.rkt` which is a source file containing the transformed interpreter, i.e., an abstract machine using the command `semt file.rkt`. The options modifying the behavior of the tool can be displayed with the command `semt --help`:

```
Usage: semt FILE [-o|--output DIR] [-i|--intermediate]
            [-d|--debug] [-t|--self-test]
  Transform an interpreter into an abstract machine.

Available options:
  FILE                     Source file with the interpreter.
  -o,--output DIR          Output directory for generated
                            files, defaults to ./out/
  -i,--intermediate        Emit executable source files for
                            each stage.
  -d,--debug               Emit labeled source files for
                            each stage.
  -t,--self-test           Run raco test on each
                            intermediate file; implies
                            --intermediate
  -h,--help                Show this help text
```

Source File Format

The tool assumes that the source file with an interpreter is a *Racket* program. An example source listing is shown in Figure A.1. The preamble (everything up to the `; begin interpreter` marker in line 4) is copied verbatim by the tool and should be used to specify *Racket*'s dialect (line 1) and import syntax definitions (line 2). Afterwards an interpreter is specified (lines 4 – 31) and it ends with another marker `; end interpreter` in line 32. Everything following the marker is again copied to the output file. In the example this space is used to define some tests for the interpreter.

Syntax

The syntax of *IDL* is given in Figure A.2. The interpreter consists of a sequence of datatype (*data-def*), record (*struct-def*) and function (*fun-def*) definitions. One of the functions must be named `main` and will serve as an entry point of the interpreter. It is required for the `main` function to have parameters annotated with types, e.g., `(def main ([Term term]) body)`. Names (*tp-name*) of datatypes and records must be unique and distinct from base types (*base-tp* and `Any`). All definitions may be mutually recursive. The term syntax is split into terms (*term*) and statements (*statements*) where the latter appear as the bodies of function definitions (both top-level and anonymous) and branches. Statements are sequences of let-bindings terminated by a regular term (see lines 16–19 of Figure A.1). Variables (*var*) and type names (*tp-name*) may consist of *ASCII* letters, decimal digits and symbols in `-+/*_?<`. Variables begin either with a lower-case letter or symbol and type names begin with an upper-case letter.

Execution

In order to run an interpreter embedded in a source file, the library with syntax definitions (`idl.rkt`) provided with this thesis must be imported. The file is located in the source tree of the project in the `interpreters/lib/` directory.

The run-time behavior of the program follows the call-by-value behavior of *Racket*, e.g.:

- `let` evaluates the bound term to a value before binding. The binding is visible only in following statements.
- Application evaluates terms left-to-right and then applies the value in function position.
- `match` evaluates the term under scrutiny and tests patterns against the value in order of their definition continuing with the first match.

```
#lang racket
(require "../lib/idl.rkt")

; begin interpreter
(def-data Term
  String            ;; variable
  {Abs String Term} ;; lambda abstraction
  {App Term Term}   ;; application
  {Unit})           ;; unit value (for testing)

(def eval (env [Term term])
  (match term
    ([String x] (env x))
    ({Abs x body}
     (fun (v) (eval (extend env x v) body)))
    ({App fn arg}
      (let fn (eval env fn))
      (let arg (eval env arg))
      (fn arg))
    ({Unit} {Unit})))

(def extend #:atomic (env k v)
  (fun #:atomic #:name Extend #:apply lookup (x)
    (match (eq? x k)
      (#t v)
      (#f (env x)))))

(def init #:atomic (x) (error "empty environment"))

(def main ([Term term])
  (eval init term))
; end interpreter

(module+ test
  (require rackunit)
  (define pgm
    {App {Abs "x" "x"} {Unit}})
  (check-equal? (main pgm) {Unit})
)
```

Figure A.1: An example of a source file

```
data-def    ::= (def-data tp-name tp-elem ...)
tp-elem     ::= tp | record
record      ::= {tp-name record-field ...}
record-field ::= tp | var | [tp var]
record-def  ::= (def-struct record)
base-tp     ::= String | Integer | Boolean
tp          ::= Any | base-tp | tp-name
fun-def     ::= (def var annot ... (arg ...) statements)
annot       ::= #:no-defun | #:atomic | #:name tp-name | #:apply var
arg         ::= var | [tp var]
const       ::= integer | string | #t | #f
statements  ::= (let var term) statements | term
term        ::= var | (fun annot ... (arg ...) statements) | (term term ...)
              | {tp-name term ...} | (match term branch ...) | (error string)
branch      ::= (pattern statements)
pattern     ::= var | const | _ | [base-tp var] | {tp-name pattern ...}
```

Figure A.2: Syntax of *IDL*

The interpreters may use the following builtin operations: `+`, `-`, `*`, `/`, `neg`, `not`, `and`, `or`, `eq?`, `<` with the usual semantics.

Annotations

- `#:apply` specifies the name for *apply* function generated for the defunctionalized function space whose member is the annotated function.
- `#:name` specifies the name for the record which will represent the annotated function after defunctionalization.
- `#:no-defun` skips defunctionalization of the annotated function (either all or none of the functions in a space must be annotated).
- `#:atomic` means that the annotated function (and calls to it) will be left in direct style during translation to CPS.

Bibliography

[1] Dana S. Scott. Data Types as Lattices. In: *SIAM J. Comput.* 5.3 (1976), pp. 522–587.

[2] Gilles Kahn. Natural Semantics. In: *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*. Ed. by Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing. Vol. 247. Lecture Notes in Computer Science. Springer, 1987, pp. 22–39.

[3] Arthur Charguéraud. Pretty-Big-Step Semantics. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 41–60.

[4] Gordon D. Plotkin. A structural approach to operational semantics. In: *J. Log. Algebraic Methods Program.* 60-61 (2004), pp. 17–139.

[5] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. Semantics Engineering with PLT Redex. MIT Press, 2009. ISBN: 978-0-262-06275-6.

[6] Casper Bach Poulsen and Peter D. Mosses. Deriving Pretty-Big-Step Semantics from Small-Step Semantics. In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 270–289.

[7] Ferdinand Vesely and Kathleen Fisher. One Step at a Time - A Functional Derivation of Small-Step Evaluators from Big-Step Counterparts. In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by Luís Caires. Vol. 11423. Lecture Notes in Computer Science. Springer, 2019, pp. 205–231.

[8] P. J. Landin. The Mechanical Evaluation of Expressions. In: *Comput. J.* 6.4 (1964), pp. 308–320.

[9] Jean-Louis Krivine. A call-by-name lambda-calculus machine. In: *High. Order Symb. Comput.* 20.3 (2007), pp. 199–207.

[10] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In: *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*. Ed. by Martin Wirsing. North-Holland, 1987, pp. 193–222.

[11] Pierre Crégut. An Abstract Machine for Lambda-Terms Normalization. In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*. ACM, 1990, pp. 333–340.

[12] Dariusz Biernacki et al. Abstracting algebraic effects. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 6:1–6:28.

[13] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In: *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*. Ed. by James Chapman and Wouter Swierstra. ACM, 2016, pp. 15–27.

[14] Olivier Danvy and Lasse R Nielsen. Refocusing in reduction semantics. In: *BRICS Report Series* 11.26 (2004).

[15] Filip Sieczkowski, Malgorzata Biernacka, and Dariusz Biernacki. Automating Derivations of Abstract Machines from Reduction Semantics: - A Generic Formalization of Refocusing in Coq. In: *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*. Ed. by Jurriaan Hage and Marco T. Morazán. Vol. 6647. Lecture Notes in Computer Science. Springer, 2010, pp. 72–88.

[16] Malgorzata Biernacka, Witold Charatonik, and Klara Zielinska. Generalized Refocusing: From Hybrid Strategies to Abstract Machines. In: *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*. Ed. by Dale Miller. Vol. 84. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 10:1–10:17.

[17] Mads Sig Ager. From Natural Semantics to Abstract Machines. In: *Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, Verona, Italy, August 26-28, 2004, Revised Selected Papers*. Ed. by Sandro Etalle. Vol. 3573. Lecture Notes in Computer Science. Springer, 2004, pp. 245–261.

[18] John Hannan and Dale Miller. From operational semantics to abstract machines. In: *Mathematical Structures in Computer Science* 2.4 (1992), pp. 415–459.

[19] Mads Sig Ager et al. A functional correspondence between evaluators and abstract machines. In: *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*. ACM, 2003, pp. 8–19.

[20] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In: *High. Order Symb. Comput.* 11.4 (1998), pp. 363–397.

[21] Mads Sig Ager et al. From interpreter to compiler and virtual machine: a functional derivation. In: *BRICS Report Series* 10.14 (2003).

[22] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. In: *Inf. Process. Lett.* 90.5 (2004), pp. 223–232.

[23] Maciej Piróg and Dariusz Biernacki. A systematic derivation of the STG machine verified in Coq. In: *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*. Ed. by Jeremy Gibbons. ACM, 2010, pp. 25–36.

[24] Dariusz Biernacki and Olivier Danvy. From Interpreter to Logic Engine by Defunctionalization. In: *Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers*. Ed. by Maurice Bruynooghe. Vol. 3018. Lecture Notes in Computer Science. Springer, 2003, pp. 143–159.

[25] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. In: *Log. Methods Comput. Sci.* 1.2 (2005).

[26] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. In: *Theor. Comput. Sci.* 342.1 (2005), pp. 149–172.

[27] Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. In: *J. Comput. Syst. Sci.* 76.5 (2010), pp. 302–323.

[28] Wojciech Jedynak, Malgorzata Biernacka, and Dariusz Biernacki. An operational foundation for the tactic language of Coq. In: *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*. Ed. by Ricardo Peña and Tom Schrijvers. ACM, 2013, pp. 25–36.

[29] David Van Horn and Matthew Might. Abstracting abstract machines. In: *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. Ed. by Paul Hudak and Stephanie Weirich. ACM, 2010, pp. 51–62.

[30] Andrew W. Appel. Compiling with Continuations. Cambridge University Press, 1992. ISBN: 0-521-41695-7.

[31] Lasse R. Nielsen. A Selective CPS Transformation. In: *Seventeenth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2001, Aarhus, Denmark, May 23-26, 2001*. Ed. by Stephen D. Brookes and Michael W. Mislove. Vol. 45. Electronic Notes in Theoretical Computer Science. Elsevier, 2001, pp. 311–331.

[32] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and Correctness of Program Transformations Based on Control-Flow Analysis. In: *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings*. Ed. by Naoki Kobayashi and Benjamin C. Pierce. Vol. 2215. Lecture Notes in Computer Science. Springer, 2001, pp. 420–447.

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

[34] Matthias Felleisen and Daniel P. Friedman. A Calculus for Assignments in Higher-Order Languages. In: *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 1987, pp. 314–325.

[35] Cormac Flanagan et al. The Essence of Compiling with Continuations. In: *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*. Ed. by Robert Cartwright. ACM, 1993, pp. 237–247.

[36] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Principles of program analysis. Springer, 1999. ISBN: 978-3-540-65410-0.

[37] Andreas Abel. „Normalization by Evaluation: Dependent Types and Impredicativity". 2013.