The Functional Correspondence Applied: An Implementation of a Semantics Transformer

(...)

Maciej Buszka

Praca magisterska

Promotor: dr hab. Dariusz Biernacki

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

9 czerwca 2020

Abstract

. . .

. . .

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 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 programs and final values in case of big-step semantics; by defining a relation allowing for step-by-step transformation of programs in case of small-step semantics; or by specifying an abstract machine with a set of states and a transition function. These semantic formats all allow for systematic definition of programming languages but differ in style and type of reasoning they allow as well as their limitations.

Another approach is to provide 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 [8] may also be classified into groups similar to the formal semantics. Starting with the most abstract and 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). Via 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. Ending with a first-order machine-like interpreters which use an explicit stack for handling control-flow of the object-language.

In his paper [8] Reynolds summarizes techniques which allow one to transform high-level definitional interpreters into lower-level ones using two well-known program transformations: transformation into continuation-passing style and defunctionalization. This connection between evaluators on different levels of abstraction has later been studied by Ager et al.[2] 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 de-

riving a correct-by-construction abstract machine given an evaluator in a diverse set of languages [2, 4, 3, 1]. 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. This algorithm has also been implemented 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 rest of this thesis is structured as following: 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 evaluate the performance of the tool and discuss related work. Appendices contain user's and developer'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. 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:

```
(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

`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 (TODO refs) semantic formats: denotational semantics, big-step operational semantics and abstract machines. These formats 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. Neverthe-

less 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. 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, 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 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. The machine consists of a set of configurations (tuples), an injection of program into initial configuration, an extraction function of result from 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 performing normal order (call-by-name) reduction of λ-calculus with de Bruijn indices. It uses two stacks: `Cont`inuation and `Env`ironment. Both of them contain `Thunk`s – not-yet-evaluated terms paired with their environment. The object-language functions are represented as `Closure`s – 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 `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 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

```
(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. The technique was first characterized and described in [2] and then later studied in context of various object-languages and their evaluators in (TODO more references). 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's) 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 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 object-language is interpreted using application in 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.

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

$$
\begin{aligned}
[\![\texttt{x}]\!]k &= (k\ \texttt{x}) \\
[\![\texttt{(fun (x y ...)\ \ e)}]\!]k &= (k\ \texttt{(fun (x y ...\ \ k')}\ [\![\texttt{e}]\!]\texttt{k'})) \\
[\![(e_1 \ldots e_n)]\!]k &= [\![e_1]\!]\ \texttt{(fun (v1)} \ldots [\![e_n]\!]k' \ldots) \\
&\quad \text{where } k' = \texttt{(fun (vn) (v1...vn}\,k)) \\
[\![\texttt{(match}\ e\,(p_1\ e_1) \ldots (p_n\ e_n)]\!]k &= [\![e]\!]\texttt{(fun (v) (case v}\ ps\texttt{)} \\
&\quad \text{where } ps = (p_1\ [\![e_1]\!]k) \ldots (p_n\ [\![e_n]\!]k) \\
[\![\texttt{(error}\ s\texttt{)}]\!]k &= \texttt{(error}\ s\texttt{)}
\end{aligned}
$$

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 ap-

```
(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

proaches to compiling first class functions away but of particular interest to us will 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 program has a unique label and that all generated names and structure labels are fresh. Whenever a top-level function is called the application

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

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

is transformed into top-level call with the sub-expressions transformed. Any other 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 ot the CEK machine [5] in Figure 2.5. It uses a stack `Cont` (which are defunctionalized continuations) to handle the control-flow and `Closure`s (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, where all functions of a particular type end up in the same bag. 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. I also chose 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). The partitioning of function spaces is done using results of control-flow analysis which approximates the runtime behavior of programs. This approach allows for functions of the same type to land in different spaces based on their usage in a program. Finally, as the transformation generates new variables and moves code around we have to keep them readable. Current implementation employs some heuristics and optional annotations to keep the names under control and inlines most of the introduced intermediate terms.

The abstract syntax of *IDL* is presented in Figure 3.1. The meta-variables x, y, z denote variables; r denote 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 one of variable, literal value, anonymous function, application,

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

Figure 3.1: Abstract syntax of *IDL*

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

In the remainder of this chapter I will describe the four stages of the transformation: translation to administrative normal form, selective translation to continuation-passing style, selective defunctionalization and let-inlining. I will also describe the algorithm used to compute the control-flow analysis.

3.1 Administrative Normal Form

The administrative normal form (ANF) [6] 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 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-continuation (i.e. a continuation in meta-language) which will be called to obtain the term for the rest of transformed input. This function 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 passed argument c when necessary. Function $[\![\cdot]\!]_s\,\cdot$ sequences

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

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

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

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

Figure 3.2: ANF transformation for *IDL*

computation of multiple expressions by creating a chain of let-bindings (using $[\cdot]_a$) and then calling the continuation with created variables.

TODO: - better characterization of the ANF - describe the algorithm - explain why this particular form

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 [7]. 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 [7], (annotated) type systems [7] and abstract abstract machines [9]. 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 summarize the derivation here but an interested reader should definitely acquaint themselves with the original work [9].

We will begin with an abstract machine for terms in A-normal form presented in Figure ???. The machine explicitly allocates memory both for values and for continuations. The environment maps variables to locations in value store and the machine keeps an address of the current continuation. Complex values (like records) also contain addresses of sub-terms rather than values themselves. This formulation ensures that the space of machine-states can be bounded by limiting store sizes. To this end we will use structural abstraction and approximate concrete values with abstract ones. Consequently, abstract stores will map addresses to sets of abstract values and continuations respectively, in order to account for their now finite domains.

3.3 Selective CPS

3.4 Selective Defunctionalization

3.5 Let Inlining

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

$$
\begin{array}{|rl|}
\hline
\langle \sigma, \rho, e, \kappa \rangle_e & \Rightarrow \varsigma \\
\hline
\langle \sigma, \rho, x, \kappa \rangle_e & \Rightarrow \langle \sigma, \rho(x), \kappa \rangle_c \\
\langle \sigma, \rho, b, \kappa \rangle_e & \Rightarrow \langle \sigma', \nu, \kappa \rangle_c \\
 & \text{where } \langle \sigma', \nu \rangle = \mathit{alloc}_v(b, \sigma) \\
\langle \sigma, \rho, \texttt{\{}r\ x \ldots\texttt{\}}, \kappa \rangle_e & \Rightarrow \langle \sigma', \nu, \kappa \rangle_c \\
 & \text{where } \langle \sigma', \nu \rangle = \mathit{alloc}_v(\texttt{\{}r\ \rho(x) \ldots\texttt{\}}, \sigma) \\
\langle \sigma, \rho, \texttt{(fun (}x \ldots\texttt{)}\ e\texttt{)}, \kappa \rangle_e & \Rightarrow \langle \sigma', \nu, \kappa \rangle_c \\
 & \text{where } \langle \sigma', \nu \rangle = \mathit{alloc}_v(\langle \rho, x \ldots, e \rangle, \sigma) \\
\langle \sigma, \rho, \texttt{(let } p\ c\ e\texttt{)}, \kappa \rangle_e & \Rightarrow \langle \sigma', \rho, c, \kappa' \rangle_e \\
 & \text{where } \langle \sigma', \kappa' \rangle = \mathit{alloc}_k(\langle \rho, p, e, \kappa \rangle, \sigma) \\
\langle \sigma, \rho, \texttt{(}x\ y \ldots\texttt{)}, \kappa \rangle_e & \Rightarrow \mathit{apply}(\sigma, \rho(x), \rho(y) \ldots) \\
\langle \sigma, \rho, \texttt{(match } x\ \texttt{(}p\ e\texttt{)}\ldots\texttt{)}, \kappa \rangle_e & \Rightarrow \mathit{match}(\sigma, \rho, \rho(x), \langle p, e \rangle \ldots) \\
\langle \sigma, \nu, \kappa \rangle_c & \Rightarrow \mathit{match}(\sigma, \rho, \nu, \kappa', \langle p, e \rangle) \\
 & \text{where } \langle \rho, p, e, \kappa' \rangle = \sigma(\kappa) \\
\hline
\mathit{apply}(\sigma, \nu, \nu' \ldots, \kappa) & = \begin{cases} \langle \sigma, \rho[(x \mapsto \nu') \ldots], e, \kappa \rangle \\ \quad \text{when } \sigma(\nu) = \langle \rho, x \ldots, e \rangle \\ \langle \sigma, \rho_i[(x \mapsto \nu') \ldots], e, \kappa \rangle \\ \quad \text{when } \sigma(\nu) = \texttt{(def } y\ \texttt{(}x \ldots\texttt{)}\ e\texttt{)} \\ \langle \sigma', \nu'', \kappa \rangle_c \quad \text{when } \sigma(\nu) = \delta \\ \quad \text{and } \langle \sigma', \nu'' \rangle = \mathit{alloc}_v(\delta(\sigma(\nu') \ldots), \sigma) \end{cases} \\
\mathit{match}(\sigma, \rho, \nu, \kappa, \langle p, e \rangle \ldots) & = \langle \sigma, \rho', e', \kappa \rangle_e \text{ where } \rho' \text{ is the environment} \\
 & \quad \text{for the first matching branch with body } e' \\
\hline
\end{array}
$$

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

Chapter 4

Evaluation

Appendix A

User's Manual

Appendix B

Developer's Manual

Bibliography

[1] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. In: *Theoretical Computer Science* 342.1 (2005). Applied Semantics: Selected Topics, pp. 149–172. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2005.06.008. URL: http://www.sciencedirect.com/science/article/pii/S0304397505003439.

[2] Mads Sig Ager et al. A Functional Correspondence between Evaluators and Abstract Machines. In: *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*. PPDP '03. Uppsala, Sweden: Association for Computing Machinery, 2003, pp. 8–19. ISBN: 1581137052. DOI: 10.1145/888251.888254. URL: https://doi.org/10.1145/888251.888254.

[3] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. In: *Logical Methods in Computer Science* 1.2 (Nov. 2005). Ed. by Philip Wadler. ISSN: 1860-5974. DOI: 10.2168/lmcs-1(2:5)2005. URL: http://dx.doi.org/10.2168/LMCS-1(2:5)2005.

[4] Dariusz Biernacki and Olivier Danvy. From Interpreter to Logic Engine by Defunctionalization. In: *Logic Based Program Synthesis and Transformation*. Ed. by Maurice Bruynooghe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 143–159. ISBN: 978-3-540-25938-1.

[5] Mattias Felleisen and D. P. Friedman. A Calculus for Assignments in Higher-Order Languages. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: Association for Computing Machinery, 1987, p. 314. ISBN: 0897912152. DOI: 10.1145/41625.41654. URL: https://doi.org/10.1145/41625.41654.

[6] Cormac Flanagan et al. The Essence of Compiling with Continuations. In: *SIGPLAN Not.* 28.6 (June 1993), pp. 237–247. ISSN: 0362-1340. DOI: 10.1145/173262.155113. URL: https://doi.org/10.1145/173262.155113.

[7] Flemming Nielson, Hanne Nielson, and Chris Hankin. Principles of Program Analysis. Jan. 1999. DOI: 10.1007/978-3-662-03811-6.

[8] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM '72. Boston, Massachusetts, USA: Association for Computing Machinery, 1972, pp. 717–740. ISBN: 9781450374927. DOI: 10.1145/800194.805852. URL: https://doi.org/10.1145/800194.805852.

[9] David Van Horn and Matthew Might. Abstracting Abstract Machines. In: *SIGPLAN Not.* 45.9 (Sept. 2010), pp. 51–62. ISSN: 0362-1340. DOI: 10.1145/1932681.1863553. URL: https://doi.org/10.1145/1932681.1863553.