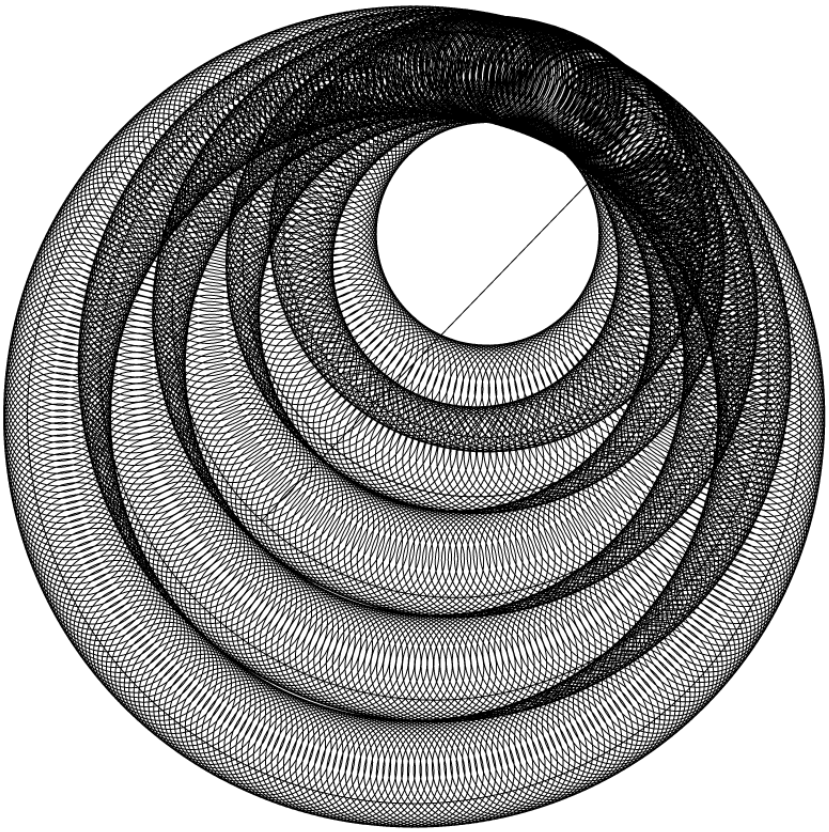
Compiling with Command Trees

Master's Thesis



Bernard Bot

Compiling with Command Trees

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bernard Bot
born in Amsterdam, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: Circles generated with JavaScript.

Source Code available on GitHub: `https://github.com/BernardBot/LamToWat`

Compiling with Command Trees

Author: Bernard Bot
Student id: 4497228

Abstract

Compilers translate high-level source code into low-level machine code. To represent source code a compiler uses a language called the intermediate representation (IR). An IR for the compilation of functional languages is continuation-passing style (CPS). It provides convenient abstractions for both data flow and control flow. However, CPS conversion is hard to write and the transformations on CPS are untyped.

In this thesis we develop an IR based on CPS using the command tree data structure. Command trees allow us to express compiler transformations typically, declaratively, and modularly. The monadic nature of command trees allows us to bind commands together in a succinct manner.

We test the usefulness of the new IR by building two versions of the LamToWat compiler that translates the lambda calculus into WebAssembly. The first version will use a CPS IR and the second version a command tree IR.

Thesis Committee:

Chair:	Prof. dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Dr. B. K. Ozkan, Faculty EEMCS, TU Delft
University Supervisor:	Dr. C. Bach Poulsen, Faculty EEMCS, TU Delft

Preface

I would like to thank my supervisor Casper Bach Poulsen for introducing me to Haskell. His guidance and our weekly meetings were essential to the completion of this thesis. I think my view on programming has been changed permanently for the better.

I would also like to thank my parents for motivating me to finish writing this document. If they had not pushed me, I would now be constructing the 30th version of the LamToWat compiler.

A special thanks goes to my roommates, who helped me release stress when it was necessary. Another special thanks goes to my friends from the library, who helped me put things in perspective.

Bernard Bot
Delft, the Netherlands
May 18, 2021

Contents

List of Figures

Chapter 1

Introduction

Continuation-passing style (CPS) is a time-tested paradigm of functional compilation [25, 2]. It bridges the gap between high-level programming languages and (abstract) machine code. Primarily used for compiling first-class functions, CPS is flexible enough to compile many other language features. CPS is a style of programming, but can also be seen as a language in itself, namely a restricted form of the lambda calculus [9, 3]. This fact gives it a solid theoretical foundation.

The language that is used by a compiler to represent source code is called an intermediate representation, as can be seen in figure 1.1. A compiler writer uses an IR to implement optimizations and translations. An IR is designed to make compilation possible and pleasant. This is what CPS provides with its records and continuations. Resulting in a convenient abstraction for data flow and control flow [5].

We will examine the following improvements on CPS in this thesis:

- Easier to write CPS conversion.
- The ability to type compiler transformations more strongly.
- A modular compiler interface for the programmer.

The data structure we will use for this examination is a command tree [22]. Our command tree based IR will have a declarative front-end and a modular structure.

To validate the usefulness of the changes to our IR we will build a small compiler in Haskell [14] for WebAssembly [29]. Haskell implements languages using data structures, which formalize the specification of a language. Our source language will be the lambda calculus. Even though the lambda calculus is a very simple functional programming language, it has sufficient abstraction to necessitate compilation to a low-level language like WebAssembly. The first-class functions of the lambda calculus will need to be translated to the second-class functions of WebAssembly. The compiler will have multiple smaller translation steps and multiple passes over the IR. The translation from the lambda calculus to our IR and the elimination of first-class functions will demonstrate the usefulness of our IR.

The contributions of this thesis are the following:

- Two version of LamToWat: a WebAssembly compiler for the lambda calculus.

$$String \xrightarrow{parse} IR_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} IR_n \xrightarrow{emit} String$$

Figure 1.1: Abstract compiler organization made up of multiple Intermediate Representation (IR) steps

- Command Trees as an IR for functional language compilation.

This thesis will continue with a chapter on the first version of the compiler based on CPS. In the third chapter we will go into some detail about command trees and show the improved version of the compiler. We will evaluate the improvements by comparing both versions of the compiler. In chapter 4 we will discuss validation and extension of our compiler. In chapter 5 we will discuss related work to support our analysis and give context to our compilation scheme.

Chapter 2

Compiling with Continuations

In this chapter we will develop the first version of the LamToWat compiler that translates the `Lam` language into the `Wat` language. Our compiler will be written in Haskell [14] and as such `Lam` and `Wat` are implemented as Haskell data types. We follow a minimal version of the approach by Appel [2]. Consequently, a simplified version of the CPS language used by Appel will serve as IR and have its own data type `Cps`.

What makes CPS favorable as an IR is that it makes control flow and data flow explicit. These features nicely represent the objective of a compiler: translating from a high-level to a low-level language by describing abstractions in finer detail. CPS uses special functions, called 'continuations', to describe more complex control flow constructs. The fact that continuations are special functions becomes essential when translating languages with first-class functions.

The purpose of building LamToWat is to examine and then improve its IR. The complexity of CPS conversion and difficulties with the untyped IR transformations will become apparent. If we can improve upon the already favorable features of CPS, we are creating a better IR to program in. Although we are not concerned with the peripherals of the compiler, an IR is not used in a vacuum. We want to examine the compiler passes that map to and from the IR as well as the ones that map to the IR itself.

LamToWat is a multipass compiler. This means that multiple passes over the IR will be made, each transforming a part of the representation. The complete compiler comprises these IR transformations with additional passes to parse lambda calculus source files and emit WebAssembly, such that it can be compiled and interpreted by the WebAssembly Binary Toolkit (WABT) [28].

Each compiler pass will have a type. A type indicates what a compiler pass does. For example, when we transform `Lam` into `Cps` with `lam2cps`, the type of this transformation is `lam2cps :: Lam -> Cps`. The double colon indicates a type declaration in Haskell. Some transformations need extra functionality, like generating fresh variable names. This functionality or effect will be handled by a helper function. For the example this will be `l2c`, which can generate fresh variables, but has a much more complicated type. We will focus on the type of the 'main' function, such as `lam2cps` when comparing the two version of LamToWat.

In this version of LamToWat the data types we will be using are `Lam`, `Cps`, and `Wat`. To read in source files we also need the data type `String`. LamToWat will be composed of the sequences of transformations in figure 2.1.

$$String \xrightarrow{parse} Lam \xrightarrow{lam2cps} Cps \xrightarrow{cps2cps} Cps \xrightarrow{cps2wat} Wat \xrightarrow{emit} String$$

Figure 2.1: LamToWat Version 1: compiler organization.

LamToWat is split into a front-end and a back-end. The front-end of the compiler is made up of parsing (`parse`), and CPS conversion (`lam2cps`). We will not discuss parsing in detail, as the topic is orthogonal to the research in this thesis. We use `parsec` [20] to build a parser for our compiler. The back-end of LamToWat consist of three transformations on the IR: closure conversion, hoisting, and emitting (`cps2wat`). In the first version of the compiler hoisting and closure conversion are combined in `cps2cps`.

The following sections will first describe the data types `Lam`, `Cps`, and `Wat` used in LamToWat with specific attention to `Cps`. We will adhere to the organization of the compiler and discuss the relevant compiler passes over these data types in the order described in figure 2.1.

2.1 Data Types

The data types of LamToWat will be implemented as Haskell data types. Haskell data types are indicated with the `data` keyword. Every data type has a name, which comes after the keyword. A number of constructors follow separated by vertical bars. Data types can be recursively defined. Type aliases indicated with the `type` keyword are also used; providing new names for existing data types.

First, we give definitions of types and data types that are used by all passes of the compiler. We define `Var`iables that indicate names of functions and values with simple `String`s. This is a design choice as it allows for nonsense programs that use undefined variables, but also for easy manipulation of variables and generation of variable names. A `Fun`ction is a triple of a function name, list of function parameter names, and a body. A `Fix`point of functions is a list of functions and a final expression.

```
type Var = String
type Fun e = (Var,[Var],e)
type Fix e = ([Fun e],e)
```

Our compiler uses a notion of `Val`ues that is incorporated in all data types. We use `LABEL`s for variables indicating function names and `VAR`s for everything else.

```
data Val
  = INT Int
  | VAR Var
  | LABEL Var
```

2.1.1 The Lam Language

The `Lam` data type represents the abstract syntax tree (AST) of the lambda calculus. The three constructors for `Lam`bda abstraction and `App`lication encompass the standard definition of the lambda calculus. `Val`ues are represented by the `Val` constructor. A constructors for `Add`ition is added to make basic arithmetic possible. The lambda calculus is one of the smallest functional languages that still has many of the interesting properties of functional languages, such as first-class functions and recursion. Moreover, the lambda calculus is well-studied [3]. Programs written in the lambda calculus will be used for testing LamToWat.

```
data Lam
  = Lam Var Lam
  | App Lam Lam
  | Add Lam Lam
  | Val Val
```

Two programs will be used as running examples throughout this thesis to show how the compilation process works. Their representation in different stages of both versions of the compiler can be found in appendix A. In the `Lam` language they are written as follows.

```
App (Lam "x" (Add (Val (VAR "x")) (Val (INT 1)))) (Val (INT 41))
```

```
App
(App
   (Lam "x" (Lam "y" (Add (Val (VAR "x")) (Val (VAR "y")))))
   (Val (INT 13)))
(Val (INT 29))
```

The first program applies an anonymous function that adds 1 to its argument `x` to the number 41. The second programs applies a nested anonymous function that adds its arguments to the numbers 13 and 29. Both programs evaluate to the number 42. We choose a program with nested functions and a program without nested functions as examples to display the compilation of this language feature separately.

LamToWat provides a parser for the `Lam` language. The accepted syntax is the same as the lambda calculus syntax in Haskell. The symbols `\` and `->` are used for lambda abstraction and a space is used for application. With this syntax the two example programs are written as follows:

```
(\ x -> x + 1) 41
```

```
(\ x -> (\ y -> x + y)) 13 29
```

2.1.2 The Cps Language

We follow the CPS language data type definition and semantics by Appel [2] with some minor adjustments. It is used as IR because it facilitates compiler transformations relevant to functional programming languages. The data type is similar to the control flow graph of a program. Control flow is modeled with functions and function calls. Data flow is modeled with records. LamToWat uses `Cps` to transform nested, higher-order functions to simple functions.

`Cps` functions are made up of a name, argument names, and a body. They are defined in mutually recursive `Fix`ed function blocks. All expressions except `App` and `Val` have a `Cps` continuation as their last argument. Since the `Add`, `Record`, and `Select` expressions produce a value, they name it before continuing. `Val` represents a final value or return statement, it does not adhere to continuation-passing style. No problems are caused by it, because we will make sure that it is the final expression of our programs. This makes it a global return of the program itself, not of a function.

```
data Cps
  = App Val [Val]
  | Val Val
  | Record [Val] Var Cps
  | Select Int Val Var Cps
  | Add Val Val Var Cps
  | Fix [Fun Cps] Cps
```

The `Cps` data type guarantees that compiler transformations can be easily implemented by enforcing the following properties:

- Functions do not return, instead the last thing a function does is call a continuation.
- Function parameters can only be values.

- All intermediate values have names.

CPS also requires all user functions to have an extra continuation argument. This requirement is not enforced by the data type, but needs to be fulfilled in order for it to be proper CPS. To illustrate how the `Cps` data type is used we will translate the first lambda calculus example into `Cps`:

```
Fix [("f", ["x"],
  Add (VAR "x") (INT 1) "r" (Val (VAR "r")))]
(App (VAR "f") [INT 41])
```

We define a function in a `Fix` block with name `f` and argument `x`. The body of this function adds `x` and 1, names the result `r`, and returns this result. Finally, we apply `f` to 41.

2.1.3 The Wat Language

The `Wat` language is a hierarchical, high-level assembly language and a simplification of WebAssembly [29]. As the final language in the compilation process it gives some guarantees about the well-formedness of the output of the compiler. The fundamental unit of code is a `Fix` made up of `Exp`.

We have chosen `Wat` as the output language of LamToWat, because it is a relatively high-level language without first-class functions. It can be easily translated to WebAssembly, but still has enough abstraction to be able to target other languages. The absence of first-class functions requires LamToWat to actually compile the `Lam` language, which does have this feature. By keeping the output of LamToWat as a Haskell data type, we can compare the semantics of the `Lam`, `Cps`, and `Wat` data types more easily.

If we look at the data type definition below, we see that `Wat` is quite similar to `Cps`. It is easy to translate from `Cps` to `Wat` after closure conversion and hosting. The `Record` and `Select` constructors are replaced with `Malloc`, `Store`, and `Load`. The `LABEL` constructor from `Val` is not used, because functions are now represented by integers serving as an index into the module's function list. `Fix` is replaced with a top-level `Fix`, ensuring that there are no nested functions anymore.

```
type Wat = Fix Exp

type Offset = Int
data Exp
  = Malloc Int Var Exp
  | Store Offset Val Val Exp
  | Load Offset Val Var Exp
  | Add Val Val Var Exp
  | App Val [Val]
  | Val Val
```

The translation of our simple example program `(\ x -> x + 1) 41` into the `Wat` language is shown below. The only real difference with `Cps` is the use of the an `INT` instead of a `LABEL` to indicate a function pointer.

```
([("f",["x"],
  Add (VAR "x") (INT 1) "r" (Val (VAR "r")))],
(App (INT 0) [INT 41]))
```

We can emit `Wat` as WebAssembly that can be compiled and interpreted by WABT. When we translate to WebAssembly our program becomes a lot longer and there are many new keywords. WebAssembly requires the programmer to declare some extra information, which

leads to a lot of syntactic noise. This information can be deduced from the previous simpler `Wat` expression, so we can emit it automatically, see the following:

```
(module
(memory 1)
(global $_p (mut i32) (i32.const 0))
(table 1 funcref)
(elem (i32.const 0) $f)
(type $_t1 (func (param i32) (result i32)))
(export "_start" (func $_start))
(func $_start  (result i32)
  (call_indirect (type $_t1) (i32.const 41) (i32.const 0))
)
(func $f (param $x i32) (result i32) (local $r i32)
  (local.set $r (i32.add (local.get $x) (i32.const 1)))
  (local.get $r)
)
)
```

The first seven lines of the printed WebAssembly program declare that there is one page of mutable global memory and a mutable heap pointer. We declare one function reference of the function `f`. This function has type `_t1`. In order to run the program we need to export the `_start` function. Following the preamble we see the entry point of our program in the form of the function `_start` and our original function `f`. Function signatures now include a result type (which is always `i32` for the programs we compile) and all locally used variables indicated with the `local` keyword. When we call a function we indicate the type of that function with the `type` keyword. To refer to a function we use its index in the function `table`. In this case we refer to the function `f` by `i32.const 0`.

2.2 Transformations

Transformations or compiler passes of LamToWat will be implemented as Haskell functions. These functions will transform the data types of the previous section. A transformation can convert a data type to a different data type, or to the same data type with certain properties. A graph representation of the example lambda programs throughout the compilation process can be found in Appendix A.1 and A.2.

2.2.1 CPS Conversion

CPS conversion will transform the `Lam` data type into the `Cps` data type and is the second transformation of LamToWat, see figure 2.1. By converting our program to continuation-passing style we make control flow and data flow explicit. In practice this means that we generate fresh variable names for intermediate values and use a metacontinuation (a continuation in the metalanguage Haskell) to indicate the order of expressions.

Appel describes a CPS conversion function that takes an extra metacontinuation as argument. It also assumes some way of generating fresh variable names, which is not further specified. In Haskell the signature of such a function would look as follows. It uses `State` for fresh variable name generation.

```
l2c :: Lam -> (Val -> State Int Cps) -> State Int Cps
```

`l2c` is split up into cases for each constructor of `Lam`. Converting `Val` is trivial: apply the metacontinuation `c` to it. To convert the anonymous function `Lam` we generate fresh variables

`f` and `k` to name the function and its continuation. We pass the function name to the meta-continuation `c`. We convert the body of the function `e`, giving it a metacontinuation which calls the continuation with the result `z`. Finally, we construct a `Fix` to contain our converted function and update continuation `c'`. The `App` and `Add` cases follow a similar approach.

```
l2c (Val v) c = c v
l2c (Lam x e) c = do
  f <- fresh "f"
  k <- fresh "k"
  c' <- c (LABEL f)
  cf <- l2c e $ \ z -> return $ App (VAR k) [z]
  return $ Fix [(f,[x,k],cf)] c'
l2c (App e1 e2) c = do
  r <- fresh "r"
  x <- fresh "x"
  c' <- c (VAR x)
  cf <- l2c e1 $ \ v1 -> l2c e2 $ \ v2 -> return $ App v1 [v2, LABEL r]
  return $ Fix [(r,[x],c')] cf
l2c (Add e1 e2) c = do
  x <- fresh "x"
  c' <- c (VAR x)
  l2c e1 $ \ v1 -> l2c e2 $ \ v2 -> return $ Add v1 v2 x c'
```

How the `l2c` function works is not obvious. We have to generate variable names, pass these to the continuation and sometimes wrap the resulting values in a `Fix`. This is quite complex, control flow should be easier to describe. For binary operators like `App` and `Add` we want to evaluate the left argument first (arbitrarily chosen order), the right argument second and finally add the two resulting values. We would like to write something like the following instead of generating a variable name and exposing the metacontinuation.

```
l2c' (Add e1 e2) = do
  v1 <- l2c' e1
  v2 <- l2c' e2
  add v1 v2
```

The case for `App` is especially complicated as we have to create a return point function to serve as continuation argument to the final application. Keeping track of continuations becomes even more non-trivial when we have multiple of them. For example when we want to have exception handlers in our source language.

2.2.2 Closure Conversion

Closure conversion transforms the `Cps` data type into the `Cps` data type where functions do not contain free variables. It is the third transformation in LamToWat, see figure 2.1. The free variables of a function are passed to the function via an extra argument. We will package a function with its free variables, this package is called a 'closure'.

There are many approaches to generating closures. We take an approach optimized for simplicity. We implement closures as records, where the first element is a function pointer (implemented as a `LABEL`) and the second element is the environment of free variables. We use effects to keep track of variables (`ask`,`local`) in scope and to hoist functions to the top-level (`tell`), which makes the type of `c2c` the following:

```
c2c :: Cps -> WriterT [Fun Cps] (Reader [Var]) Cps
```

The cases for `Fix` and `App` are at the heart of the transformation. For the other constructors we simply update the environment with their assigned names.

When we encounter a `Fix` we ask what variables are in scope. For each function we add an extra first argument `_closure`: the closure record. We prefix the function body with selecting and naming all variables in scope that do not have the same name as any of the function's arguments. We open the closure by selecting the environment from the closure record and naming it `_env`.

When we encounter an `App` we create a record with all variables in scope and call it `_env`. We pack it with all function pointers to create closures and rename them by prefixing an underscore. There are now two cases for application: closures and function pointers. To apply a function pointer we create a closure and add it as the first argument. To apply a closure we select the closure's first element and apply that to the closure and the closure's original arguments.

```
c2c (Fix fs e) = do
  fs' <- mapM funClos fs
  tell fs'
  c2c e
  where funClos (name,args,body) = do
          nv <- ask
          body' <- local (++args) (c2c body)
          return $
            ( name
            , "_closure" : args
            , Select 1 (VAR "_closure") "_env" $
              foldr (openClos args) body' (zip [0..] nv)
            )

        openClos args (i,x) =
          if x `elem` args then id else Select i (VAR "_env") x

c2c (App fun args) = do
  nv <- ask
  return $
    Record (map VAR nv) "_env" $
    foldr mkClos appClos args
  where appClos = case fun of
          LABEL fp -> let cl = '_' : fp in
                        Record [LABEL fp,VAR "_env"] cl $
                        App (LABEL fp) (VAR cl : args')

          VAR cl   -> let fp = '_' : cl in
                        Select 0 (VAR cl) fp $
                        App (VAR fp) (VAR cl : args')

        mkClos (LABEL x) = Record [LABEL x, VAR "_env"] ('_' : x)
        mkClos _ = id

        args' = map rename args

        rename (LABEL x) = VAR $ '_' : x
        rename v = v

c2c (Record vs x e) = withvar x e $ Record vs
```

```
c2c (Select i v x e) = withvar x e $ Select i v
c2c (Add v1 v2 x e) = withvar x e $ Add v1 v2
c2c (Val v) = return $ Val v

withvar x e op = do
  e' <- local (++[x]) (c2c e)
  return $ op x e'
```

The main defect of `cps2cps` is that it gives no indication of the fact that our `Cps` expressions no longer contain `Fix` expressions. When we encounter a `Fix`, we use the `tell` function to write all functions to a list. It is obvious from the implementation that they are no longer present, however, the type `cps2cps :: Cps -> Cps` of the transformation gives no recognition of this fact. If we were to pass a `Cps` expression to the next pass of the compiler that has nested `Fix`es, it would cause an error.

2.2.3 Emitting

Emitting transforms the `Cps` data type into the `Wat` data type. We create a list of function names `ns` to map function names to indices. The most interesting parts of the transformation are for the `Record` case and the `LABEL` case. `Record`s are transformed into a combination of `Malloc` and `Store`, because `Wat` does not support such a high level abstraction as records, but just has simple heap operations. `LABEL`s are mapped to their index in the list of function names, transforming function labels into function pointers.

This transformation could benefit from the reuse of constructors. We see that `App` is transformed to its `Wat` counterpart `App`. We could just leave these parts of the data type untouched and only show the essence of the transformation, it would benefit both the reader and the writer of this code. What the transformation does would be more explicit and less code naturally leads to a smaller number of errors.

```
cps2wat :: Cps -> Wat
cps2wat (C.Fix fs e) = (map (fmap c2w) fs,c2w e)
    where ns = map (\ (f,as,b) -> f) fs

          c2w (C.Val v)          = W.Val (v2v v)
          c2w (C.App v vs)       = W.App (v2v v) (map v2v vs)
          c2w (C.Add v1 v2 x e)  = W.Add (v2v v1) (v2v v2) x (c2w e)
          c2w (C.Select i v x e) = W.Load i (v2v v) x (c2w e)
          c2w (C.Record vs x e)  =
            W.Malloc (length vs) x $
            foldr (\ (i,v) -> W.Store i (VAR x) (v2v v))
              (c2w e) (zip [0..] vs)

          v2v (LABEL x) = INT $ fromJust $ x `elemIndex` ns
          v2v v         = v
```

In order to run our converted expression we need WABT and indicate where the `wat2wasm` and `wasm-interp` binaries are located. We can then run the command from within a interactive Haskell session with `shake` [24]. A `ghci` session where we would run a converted lambda calculus expression would look as follows:

```
> putStrLn $ emit $ cps2wat $ cps2cps $ lam2cps $ parse "(\\ x -> x + 1) 41"
(module
...
```

```
> Wat.emitRun $ cps2wat $ cps2cps $ lam2cps $ parse "(\\ x -> x + 1) 41"
_start() => i32:42
```

2.3 Discussion

In this chapter we have shown how to build a compiler that translates `Lam` into `Wat`. We used three data types: `Lam`, `Cps`, and `Wat`. By applying the transformations `parse`, `lam2cps`, `cps2cps`, and `cps2wat` to a well formatted lambda calculus string, we obtain a low-level language output that can be printed as WebAssembly. By looking at two example lambda expressions, we tracked the state of our program throughout the compilation process, see Appendix A.1 and A.2.

We identified three problems with the transformations on the data types and the data types itself. These are summarized as follows:

- Complex specification of control flow in `lam2cps`

  `lam2cps` is itself written in Continuation-Passing Style. It requires us to expose the metacontinuation. This leads to a confusing specification of control flow where the programmer needs to constantly switch between continuation and metacontinuation.

- No types to indicate change of `Cps` after `cps2cps`

  Closure conversion has type `cps2cps :: Cps -> Cps`. The `Cps` data type is free to contain nested `Fix` expressions. We want to guarantee that functions are no longer nested. All function definitions should be contained in a single, top-level `Fix`. The bodies of the `Fix` should be made up of expressions containing only addition, records, and application.

  Alternatively, we could write separate data types for each transformation. These could indicate the types we want. However, this would lead to a lot of duplicate code, as mentioned in the next problem. The number of the extra lines of source code is calculated as follows: multiply the number of lines of your original data type (in our case `Cps`) with the number of transformations you want to perform. We see immediately that this leads to a lot of lines of code for a larger number of transformations.

- Duplicate constructors in `Cps` and `Wat`

  If we look at both data types we see that addition and application constructors match one-to-one. We would like to only transform `Record` expressions into `Malloc` expressions.

In the next chapter we will try to alleviate these shortcomings of the first version of our compiler by proposing a new data type: command trees.

Chapter 3

Compiling with Command Trees

In this chapter we propose command trees as an improvement upon the `Cps` data type of the previous chapter. The new compiler uses the `Tree` and `Tps` data type as IR. These data types are modular in the commands they use and have an internalized notion of control flow. This allows us to solve the problems from section 2.3: complex control flow specification, absence of indicative type changes, and duplicate constructors.

Compiler passes on the new IR will have a type that looks like:

```
pass :: Tps cmd Val -> Tps cmd' Val
```

The differences between `cmd` and `cmd'` indicate what a pass does.

If we look at the transformations in figure 3.1, we notice that our compiler consist of more transformations than before. We also note the reuse of the `Lam` and `Wat` data types, which are the same as in the previous version of LamToWat. Our notion of `Val`ues remains the same too. We combine different commands by using open unions ($+$ in the figure, and `:+:` in code). The increase in compiler steps does mean an increase in complexity, but also an increase in explicitness and declarativity. The complexity was present in the previous version of LamToWat, but remained hidden. The new type of `Tps` gives the programmer information about the transformations.

`Tree` is used in the front-end of the compiler, while `Tps` is used in the back-end. This chapter is structured in the same order as the previous chapter: we will first discuss the new `Tree`

$$
\begin{aligned}
&String \xrightarrow{parse} \\
&Lam \xrightarrow{lam2tree} \\
&Tree(Comp + Fix + Base)Val \xrightarrow{tree2tps} \\
&Tps(Fix + Base)Val \xrightarrow{hClos} \\
&Tps(Fix + Record + Base)Val \xrightarrow{hRecord} \\
&Tps(Fix + Malloc + Base)Val \xrightarrow{hFix} \\
&Fix(Tps(Malloc + Base)Val) \xrightarrow{tps2wat} \\
&Wat \xrightarrow{emit} \\
&String
\end{aligned}
$$

Figure 3.1: LamToWat Version 2: compiler organization. Transformations are sequenced up to reordering of commands, see 3.1 and 3.3.

and `Tps` data types and then examine their transformations. A graph representation of the example lambda programs throughout the compilation process can be found in Appendix A.3 and A.4. Finally, we will suggest command tree improvements and discuss how command trees have solved the problems of the first version of LamToWat.

3.1 Command Trees

Command trees are a data type used to sequence commands. A well known data type that is also capable of sequencing is a list. Command trees improve upon lists by adding subcontinuations and by providing the ability to use the result of our command in the commands after that. Command trees can be easily sequenced using Haskell's `do`-notation.

The meaning of a command is something that is left to the programmer [21, 31]. What a command does is implemented later in a function called a handler. Command trees are used to model effects in denotational semantics. We can write handlers for effects to create an interpreter. In this chapter we try to extend this approach to writing a compiler. Commands can be used in a modular manner by using open unions. We give two derivations of command trees in section 5.1 and section 5.2 and discuss the role of command trees in (modular) denotational semantics in more detail in section 5.3.

The structure and monadic nature of command trees have already been discussed in Compilers for Free [22]. A modified version of the discussion will be restated in this chapter for completeness. A command tree consists of two constructors: `Leaf` and `Node`. `Leaf a` is the smallest form of command tree that exists and simply returns the value `a`. A `Node cmd ks k` command tree is somewhat more complex and is made up of three parts:

- A command `cmd` that may have a signature,
- A list of subcontinuations `ks`,
- An optional continuation (also called join-point) `k`.

We will call the data type `Tree` a semantic command tree and `Tps` a syntactic command tree.

Semantic command trees are well suited for defining the initial translation into CPS. However, the abstract nature of semantic command trees prevents us from defining closure conversion without escaping from its abstraction. The fundamental problem is that continuations are defined as functions in the metalanguage instead of syntactic constructs in the IR itself. This prevents us from doing free variable analysis, which is at the heart of closure conversion.

Syntactic command trees have, as their name implies, syntactic representation of name binding in their constructors. They are specific to the notion of `Values` in `Cps`. We require a function to translate from our semantic command trees to syntactic command trees in order to compile our lambda calculus source code. Because syntactic and semantic command trees represent the same piece of source code, we conjecture that a function that translates syntactic command trees back into semantic command trees must also exist. We have not implemented such a function yet, see section 3.3.

Command trees are implemented using Haskell's Generalized Algebraic Data Type (GADT). A GADT gives us the power to add more types to our IR. The notation of a GADT is somewhat different than the notation of a normal data type. After the `data` keyword we specify a name and type parameters, a `where` keyword follows. Then the different constructors of the GADT are declared. The constructors are given a name followed by `::`. The types of the arguments follow, separated by `->`. Finally, we declare the constructor with its type parameters, which can relate to the types of the arguments.

Command trees are defined by a GADT as follows:

```
-- semantic command tree
data Tree sig a where
  Leaf :: a -> Tree sig a
  Node :: sig n b p r q ->
          Vec n (p -> Tree sig r) ->
          Option b (q -> Tree sig a) ->
          Tree sig a
-- syntactic command tree
data Tps (sig :: Sig) a where
  Leaf :: a -> Tps sig a
  Node :: sig n b p r q ->
          Vec n (Tps sig Val) ->
          Option b (Var, Tps sig a) ->
          Tps sig a
```

A command tree does not know what set of commands is used. However, it does encode strong type constraints on its subcontinuations and continuation. To enforce these constraints we use a signature `sig`. This way a command tells the command tree what comes after itself. More formally a signature has type:

```
type Sig = Nat -> Bool -> * -> * -> * -> *
```

A signature instance `sig n b p r q` tells us the following:

- The tag of the command and its enclosed parameters `sig`.
- The number of subcontinuations `n`.
- If the command has a continuation `b`.
- The argument `p` and return type `r` of the subcontinuations.
- The argument type `q` of the continuation.

Notice that `Tps` does not use all signature information. `Tps` is specialized to mirror the structure and types of `Cps` and thus only uses the `n` and `b` part of the signature.

Without commands we can not do anything with our command trees. We will use a set of commands that reflect the constructors from the `Cps` data type. We also define a number of extra commands that will help us compile.

```
data Base :: Sig where
  Add    :: Val -> Val ->        Base Z     True  Void Void Val
  App    :: Val -> [Val] ->      Base Z     False Void Void Val

data Fix :: Sig where
  Fix    :: Vec n (Var,[Var]) -> Fix n      True  ()   Val  ()

data Comp :: Sig where
  GetK   :: Var ->               Comp Z     True  Val  Val  Val
  SetK   :: Var -> Val ->        Comp Z     True  Val  Val  ()
  Block  ::                      Comp (S Z) True  ()   Val  Val
  Fresh  :: Var ->               Comp Z     True  Void Void Var

data Record :: Sig where
  Record :: [Val] ->             Record Z   True  Void Void Val
  Select :: Int -> Val ->        Record Z   True  Void Void Val
```

```
data Malloc :: Sig where
  Malloc :: Int ->               Malloc Z   True  Void Void Val
  Load   :: Int -> Val ->        Malloc Z   True  Void Void Val
  Store  :: Int -> Val -> Val -> Malloc Z   True  Void Void ()

data Empty :: Sig where
```

If we look at the definition of commands above, we can see that our original constructors `Add`, `App`, `Fix` from the `Cps` data type are here. The `Val` constructor will be modeled by the `Leaf` constructor. The `Val` type is the same as in `Cps`. The four compilation commands represent a continuation store (`GetK`,`SetK`), command concatenation (`Block`), and fresh variable name generation (`Fresh`). We also have the `Malloc` commands to represent the heap operations of `Wat`.

We take a look at the signatures of the commands to see how they are structured. The `Block` and `Fix` command are the only ones that have subcontinuations. The `Void` type indicates that there are no subcontinuations. `Fix` has a subcontinuation for every function definition. This is required by the constructor itself as the natural number `n` appears in both the function name with arguments and subcontinuations. The `App` command does not have a continuation, which will lead to some trouble when trying to concatenate it with other commands. The return type of commands indicates that commands will bind to a variable `Val`, or only produce a side-effect `()`.

The `Empty` command is special because it does not have any members. It is used to write transformations generically. For example, we define a transformation of a command to another command as the following function:

```
foo :: Tps (cmd :+: rest) Val -> Tps (cmd' :+: rest) Val
```

This transformation should also work if the only command in the tree is `cmd`. The tree would have type `Tps cmd Val`, but this does not match because there is no `rest`. This is where the `Empty` command takes the place of `rest`:

```
foo :: Tps (cmd :+: Empty) Val -> Tps (cmd' :+: Empty) Val
```

The `rest` problem is caused by the way we have implemented open unions in Haskell, see subsection 3.1 and section 3.3.

Let's see how our example `(\ x -> x + 1) 41)` translates to a semantic command tree. The translation is very similar to the `Cps` translation. We use a `Fix` command to create a function `f` with argument `x`. The body of this function adds `x` and the number one `1`. The continuation is a leaf node, which serves the same purpose as the `Val` constructor. The function name and arguments are separated by the function body, which is a vector of thunks (functions that take `()` as argument). The continuation of our function node is an application node. Here we apply the function `f` to the argument `41`.

```
Node (L (Fix (("f", ["x"]) ::: Nil))) ((\ () ->
  Node (R (Add (VAR "x") (INT 1)) Nil (Some (\ n -> Leaf n)))) ::: Nil) (Some (\ () ->
Node (R (App (VAR "f") [INT 41])) Nil None))
```

The syntactic syntax is similar to its semantic counterpart. The main difference is that function variables have been replaced with strings. In this case we use the empty string `""` to represents continuations that take `()` as an argument.

```
Node (L (Fix (("f", ["x"]) ::: Nil))) ((
  Node (R (Add (VAR "x") (INT 1))) Nil (Some ("n", Leaf (VAR "n")))) ::: Nil) (Some ("",
Node (R (App (VAR "f") [INT 41])) Nil None))
```

Open Unions

In order to compile `Lam` into `Wat` we will have to make use of all our command modules. We will combine our commands using an open union or extensible sum data type. An open union can be viewed as a list of data types. More precisely, it is a binary tree which has data types as leaf nodes. `:+:` is right-associative and has two constructors `L` and `R`, which inject a data type into the left or right side of the tree, respectively [16]. Note that we can nest instances of open unions to create open unions. Open unions make our command tree modular, because we can add new commands to an existing union to represent new language features.

```
data (:+:) :: Sig -> Sig -> Sig where
  L :: sigl n b p r q -> (sigl :+: sigr) n b p r q
  R :: sigr n b p r q -> (sigl :+: sigr) n b p r q

class (sub :: Sig) :<: (sup :: Sig) where
  inj :: sub n b p r q -> sup n b p r q
```

The typeclass `:<:` allows us to inject data types into an open union automatically. We use smart constructors to mitigate the syntactic overhead of injecting [16] even further. For example to lift our `Add` command into the command tree we can define the following function. The constraint `Base :<: cmd` ensures that `Add` is located somewhere in the commands of the command tree. We will rarely use the original commands and mostly use their smart constructors when writing our tree transformations.

```
add :: Base :<: cmd => Val -> Val -> Var -> Tps cmd ()
add v1 v2 x = liftT (inj (Add v1 v2)) Nil x (Leaf ())
```

3.2 Tree Transformations

Transformations for the new version of LamToWat will be implemented as Haskell functions. We will program mostly using `do`-notation. A graph representation of the example lambda programs throughout the compilation process can be found in Appendix A.3 and A.4.

3.2.1 CPS Conversion

Using `Tree`, we can define an improved CPS conversion. The function is easier to read and write. We no longer have a metacontinuation hidden inside a continuation monad. This simplifies the notation significantly. We still operate in a monad, however, this monad is the command tree. The order of our listed operations matches the order of our final program more closely. There are some details that spoil the declarativity of our conversion somewhat. The advantages and disadvantages of our new approach become clear when we examine the conversion of a lambda abstraction in comparison to the one in the previous version of LamToWat.

```
lam2tree :: Lam -> Tree (Comp :+: Fix :+: Base) Val
lam2tree (Val v) = return v
lam2tree (Lam x e) = do
  f <- fresh "f"
  k <- fresh "k"
  fix ((f,[x,k],do
            v <- lam2tree e
            app (VAR k) [v])
       ::: Nil)
```

```
  return (LABEL f)
lam2tree (App e1 e2) = block (do
  v1 <- lam2tree e1
  v2 <- lam2tree e2
  k <- getk "_nxt"
  app v1 [v2,k])
lam2tree (Add e1 e2) = do
  v1 <- lam2tree e1
  v2 <- lam2tree e2
  add v1 v2
```

We take a look at the conversion of `Lam`. We generate a fresh function variable `f` and continuation variable `k`. We use these variables to create a function with name `f` that has as argument the original variable and a continuation named `k`. The body of the function will be the converted original body and a final statement that applies the continuation to the resulting variable. Finally, we return a `LABEL` with the function name.

3.2.2 Tree Compilation

The `Tree` that is output by `lam2tree` contains commands that represent effects. We will need to handle these commands and instantiate name binding commands with generated variables before we can closure convert. This is necessary in order to perform variable analysis. The transformation is called `tree2tps` and is performed using effects for generating fresh variable names (`fresh`) and accessing and updating an environment that associates `Vars` with `Vals` (`ask`,`local`). The type of `t2t` becomes:

```
t2t :: Tree (Comp :+: Fix :+: Base) Val ->
       StateT Int (Reader [(Var,Val)]) Tps (Fix :+: Base :+: Empty) Val
```

Notice that we add the `Empty` command to the end of the signature of `Tps`. `t2t` itself is defined as follows:

```
t2t (Leaf x) = return (done x)
t2t (Node (R (R (Add v1 v2))) Nil (Some k)) = do
  x <- fresh "x"
  k' <- t2t (k (VAR x))
  return (add v1 v2 x k')
t2t (Node (R (R (App v vs))) Nil None) =
  return (app v vs)
t2t (Node (R (L (Fix fxs))) bs (Some k)) = do
  bs' <- mapM (\ b -> t2t (b ())) bs
  k' <- t2t (k ())
  return (fix' fxs bs' k')
t2t (Node (L (SetK x v)) Nil (Some k)) =
  local ((x,v):) (t2t (k ()))
t2t (Node (L (GetK x)) Nil (Some k)) = do
  nv <- ask
  case lookup x nv of
    Just v -> t2t (k v)
    Nothing -> error (x ++ " is not in env " ++ show nv)
t2t (Node (L Block) (b ::: Nil) (Some k)) = do
  r <- fresh "r"
  x <- fresh "x"
  b' <- local
```

```
    (("_nxt",LABEL r):)
    (t2t (do v <- b ()
             T.app (LABEL r) [v]))
  k' <- t2t (k (VAR x))
  return (fix' ((r,[x]) ::: Nil) (k' ::: Nil) b')
t2t (Node (L (Fresh x)) Nil (Some k)) = do
  f <- fresh x
  t2t (k f)
```

The compilation of the `Fresh` command seems trivial, because we use the helper function `fresh`. This helper function should not be confused by the sugared version of the `Fresh` command. This function updates the state and returns a fresh variable (in this case a string).

The compilation of `Add` shows the instantiation of variables in the metalanguage with variables in the syntactic command trees. We generate a fresh variable `x`, pass it to the continuation and compile the continuation, and finally plug the result into the syntactic command tree.

The `SetK` and `GetK` commands update and fetch named continuations. In our case there is only a continuation that is named `_nxt`. The `Block` command tells us to compile the continuation `k` by passing it `VAR x` and wrap it in a continuation function with name `r` and argument `x`. We set the continuation `_nxt` to the function label `LABEL r`. We extend the body of the block with a final application of the continuation function `r` to the result `v` and compile with the updated continuation list. Finally, we create the continuation function and give it the compiled body `b'` as continuation.

After tree compilation we obtain a `Tps` with the commands `Fix` and `Base`. We have done two transformations to obtain the same result as with `Cps`. However, these transformations are significantly easier to write and the conversion from `Tree` to `Tps` only has to be written once if we keep the command set that `Tree` has now.

3.2.3 Closure Conversion

Now that we have eliminated the `Comp` commands we can closure convert our syntactic command tree. We will follow the same approach as in the previous version of LamToWat: collect names of expression and use these to construct records. We will make the assumption that the only place where expressions are named (and thus our environment is extended) is in the join-point of a node and in functions. We do not hoist our function definitions to the top level immediately. We now have an extra transformation for this.

Separating the transformations also gives as a chance to better describe the type of the command tree before and after. The modular open unions of commands allows us to do so without having to write completely new data types which are mostly the same. Moreover, the transformations described in this section are modular: we can reuse them for different unions of commands.

The transformation only changes the `Fix` and `App` nodes. The other nodes simply extend the environment with their binding variable. The environment effect is now implemented as an extra argument to `hCl`.

```
hCl :: [Var] ->
       Tps            (Fix :+: Base :+: cmd) Val ->
       Tps (Record :+: Fix :+: Base :+: cmd) Val
hCl nv (Node (L (Fix fxs)) bs (Some (_,k))) =
  fix' (mapV addArg fxs)
       (zipWithV funClos fxs bs)
       (hCl nv k)
  where addArg (name,args) = (name,"_closure":args)
```

```
        funClos (name,args) body = do
          select_ 1 (VAR "_closure") "_env"
          zipWithM_ (openClos args) [0..] nv
          hCl (nv++args) body

        openClos args i x =
          if x `elem` args then return () else select_ i (VAR "_env") x

hCl nv (Node (R (L (App fun args))) Nil None) = do
  record_ (map VAR nv) "_env"
  args' <- mapM mkClos args

  case fun of
    LABEL fp -> let cl = '_' : fp in do
      record_ [LABEL fp,VAR "_env"] cl
      app (LABEL fp) (VAR cl : args')

    VAR cl -> let fp = '_' : cl in do
      select_ 0 (VAR cl) fp
      app (VAR fp) (VAR cl : args')

  where mkClos (LABEL x) = let _x = '_' : x in do
            record_ [LABEL x,VAR "_env"] _x
            return $ VAR _x
        mkClos v = return v

hCl nv (Leaf v) = Leaf v
hCl nv (Node cmd ks k) =
  Node (R cmd)
    (fmap (hCl nv) ks)
    (fmap (\ (x,k) -> (x,hCl (extendnv nv x) k)) k)
  where extendnv nv "" = nv
        extendnv nv x = nv ++ [x]
```

Before we hoist our function definitions to the top level we can transform our `Record` commands into `Malloc` commands. Here we see that we can fix our shortcoming of repeated constructors quite easily with command trees. We can focus on a particular command and translate it into its lower-level counterpart. In this case only `Record` as truly translated as `Select` and `Load` have a one-to-one mapping.

```
hRecord :: Tps (Record :+: cmd) Val -> Tps (Malloc :+: cmd) Val
hRecord (Node (L (Record vs)) Nil (Some (x,k))) = do
  malloc_ (length vs) x
  zipWithM_ (\ i -> store_ i (VAR x)) [0..] vs
  hRecord k

hRecord (Node (L (Select i v)) Nil (Some (x,k))) =
  load i v x (hRecord k)

hRecord (Leaf v) = Leaf v
hRecord (Node (R cmd) ks k) =
  Node (R cmd)
```

```
    (fmap hRecord ks)
    (fmap (fmap hRecord) k)
```

Hoisting is done with the `hFix` function. We will need to be able to open the join-point of a `Node` of our command tree in order to be able to hoist, because it is a non-local transformation. A non-local transformation affects the entire tree. In the case of hoisting we are chopping up the tree into individual commands, separating the `Fix` commands and putting them into a list, and glueing the other commands back together to form a new tree. The type of `hFix` clearly indicates that functions no longer contain other functions.

```
hFix :: Tps (Fix :+: cmd) Val -> T.Fix (Tps cmd Val)
hFix (Leaf v) = ([],Leaf v)
hFix (Node (R cmd) ks k) = case k of

  Some (x,k) -> (fs++fs',Node cmd ks' (Some (x,k')))
    where (fs,k') = hFix k

  None -> (fs',Node cmd ks' None)

  where ks' = mapV (snd . hFix) ks
        fs' = concatMap (fst . hFix) $ toList ks

hFix (Node (L (Fix fxs)) bs (Some ("",k))) = (fs'++fs,k')
  where fs' = concat $ zipWith hFun (toList fxs) (toList bs)
        (fs,k') = hFix k

        hFun (f,as) b = (f,as,b') : fs
          where (fs,b') = hFix b
```

3.2.4 Emitting

The emit step now becomes even more trivial, as we have also eliminated records from our command tree and replaced them with malloc commands. We include this step for completeness and testing purposes. We could use the command tree output by `hFix` to generate WebAssembly code. The mapping is one-to-one for expressions: every remaining tree command has a `Wat` counterpart. However, the transformation of `Val`s does need to change labels into integers.

```
tps2wat :: WatTps -> Wat
tps2wat (fs,e) = (map (fmap t2w) fs,t2w e)
  where ns = map (\ (f,as,b) -> f) fs

        t2w (Leaf v) = Val (v2v v)
        t2w (Node cmd ks k) = case (cmd,ks,k) of
          ((L (T.Malloc i)),      Nil, (Some (x,k))) -> Malloc i x (t2w k)
          ((L (T.Load i v)),      Nil, (Some (x,k))) -> Load i (v2v v) x (t2w k)
          ((L (T.Store i s t)),   Nil, (Some (_,k))) -> Store i (v2v s) (v2v t) (t2w k)
          ((R (L (T.Add v1 v2))), Nil, (Some (x,k))) -> Add (v2v v1) (v2v v2) x (t2w k)
          ((R (L (T.App v vs))),  Nil, None)         -> App (v2v v) (map v2v vs)

        v2v (LABEL x) = INT $ fromJust $ x `elemIndex` ns
        v2v v         = v
```

A `ghci` session where we would run a converted lambda calculus expression would look as follows:

```
> putStrLn $ emit $ tps2wat $ hFix $ swapTps $ hRecord $
    hClos $ tree2tps $ lam2tree $ parse "(\\ x -> x + 1) 41"
(module
...

> Wat.emitRun $ tps2wat $ hFix $ swapTps $ hRecord $
     hClos $ tree2tps $ lam2tree $ parse "(\\ x -> x + 1) 41"
_start() => i32:42
```

3.3 Command Tree Improvements

In this section we will explore the design space for command trees and discuss the shortcomings of the second version of LamToWat. During the development of the LamToWat compiler a number of different command trees were examined to see if they could provide us with a replacement for `Cps`. We will discuss some of the relevant features here.

Although command trees provide a useful abstraction for language implementers, it does require knowledge of the block model for control flow. Command trees help the programmer somewhat by providing metacontinuation store, which can be manipulated with the `SetK` and `GetK` commands. The language implementer will still need to wrap certain parts of code inside a block and fetch the right continuation at the right point. The responsibility is now put on the compiler writer, who has to compile the semantic command tree into a syntactic command tree.

In order to check intermediate results of the compiler after the initial CPS conversion, we would like to have an interpreter for `Tree`. We can now only print `Tree`, but we would like to map `Tree` to a common domain. The abstract nature of `Tree` is the main problem when writing an interpreter for it that is modular in the set of commands. A function called `tps2tree` that transforms `Tps` back into `Tree` for the right set of commands would serve a similar purpose. This would show the isomorphism between the two and give another check of the compilation process.

Our command trees have some types, but we would like our types to do even more. For example we want a type to indicate that an expression is closed after closure conversion, i.e., it does not have any free variables. Transforming syntactic command trees into semantic ones would work as a sort of type checking function. Typing closure conversion has been studied in Haskell and other languages [13, 8, 18]. We tried to implement something similar in LamToWat but found Haskell's type system uncooperative [17].

The implementation of open unions we have used for making `Tps` modular can be improved. There is still some extra work required of the programmer. Extensible sums should behave like sets, but are implemented as binary trees. This means that the order of commands matters, e.g., `A :+: B` is not the same as `B :+: A` in the eyes of the Haskell type system. To mitigate this we can write helper functions that transform the structure of our extensible sums. There are a number of other implementations of open unions in the Haskell language which may provide the functionality we require [11, 19]. Haskell's typeclasses could be used to derive the necessary operations on open unions.

3.4 Discussion

In this chapter we have shown how the three shortcomings of our original compiler are eliminated by using command trees as IR. Each of the shortcomings addressed at the end of the

previous chapter in section 2.3 is alleviated as follows:

- CPS conversion becomes easier to specify by using blocks in `lam2tree`
- The type of the output of `hFix` indicates that functions are no longer nested
- `hRecord` only transforms records and thus removes duplicate constructors

We separated `Cps` into two command trees: semantic and syntactic. Semantic command trees give us the power to write a declarative CPS conversion function, improving the front-end of LamToWat. Syntactic command trees specific to `Cps` allow the programmer to write modular, declarative transformations in the back-end of LamToWat without losing the ability to do variable analysis.

We can combine commands to create a modular approach to compilation. With the help of smart constructors and destructors we reduce syntactic overhead. More transformations are performed on the command tree than on `Cps`, because we have to handle effects ourselves. Although it requires a little more effort on behalf of the programmer, it also provides a method to make transformations explicit and declarative. In the next chapter we will discuss the how we validated the performance of the compiler.

Chapter 4

Compiler Validation and Extension

In this section we will discuss how we validated that our compiler worked and the performance of the transformations and command trees. We will also address some of the compiler transformations we did not implement because they were out of scope of this thesis. The online repository with the source code for LamToWat can be found on GitHub: `https://github.com/BernardBot/LamToWat`.

4.1 Testing

In order to test if all transformations were performed correctly we have written interpreters for the `Lam`, `Cps`, `Wat`, and `Tps` languages/data types. This allowed us to test programs at different steps of the compilation process. We compare the interpreter's results and check that they are the same. Of course this does not guarantee equality between programs. We could obtain the expected result by a wrong calculation, or by using other effects.

To automate the testing process we use Cabal [27], which allows us to create a test suite. By simply executing the `cabal test` command we run all automated tests. A folder of lambda calculus source files that are used for testing is provided in the projects repository.

4.2 Performance

How do command trees perform in comparison with monad transformers? The paper "Freer monads, more extensible effects" [15] compares the `mtl` library with their own extensible effects library based on the freer monad. Their results show that algebraic effects outperform monad transformers when we nest multiple effects. Since the Haskell compiler GHC has specific optimizations for the `mtl` library and especially the `State` monad, monad transformers are sometimes faster for single effects.

In order to compile command trees we have used separate handlers (`hClos`,`hRecord`,`hFix`). This is quite inefficient, because we have to build intermediate trees and traverse the entire data structure for each handler. Instead, we can fuse [30] a sequence of handlers and remove both these performance pain points. Fusion is not yet implemented for the current command trees, but may lead to a significant performance gain.

While testing both versions of LamToWat no noticeable difference was observed for both compilation and execution time of the generated programs. Since programs were comparatively small, this does not give a good indication of how command trees would perform on larger bodies of code.

4.3 Other Transformations

Our compiler is a very simplified version of the compiler described in Compiling with Continuations (corr. version) [2]. Appel discusses a number of other transformations that improve the performance of the generated code and compile other language features:

- Closure Optimization

  Creating efficient closures is not trivial. LamToWat makes simple but inefficient closures. In order to create closures that are optimized for speed or memory usage we will need to perform extended variable analysis. We could create a command that is a special form of `Fix` that carries this variable information with it. We can then implement two transformations: one that does the analysis and one that creates the better closures. The syntactic nature of `Tps` allows us to do this optimization.

- Compilation of pattern matching

  Pattern matching is a feature that benefits many functional languages. To compile pattern matching we can use `switch` expressions. `switch` expressions take a value and a list of expressions. We would need to add a switch expression to `Cps` in the first version of LamToWat and a switch command to both `Tree` and `Tps`. Our source language would have `case` statements. These would need to be compiled to `switch` expressions. This translation can be optimized by way of a decision-tree algorithm.

- Inlining functions

  The inlining of functions substitutes a function body for a function call. This increases performance of a program. Substitution of function bodies may be problematic if not performed correctly, because it may lead to a large increase in program size. To implement this feature in LamToWat we could create a new command that identifies function bodies as candidates for inline expansion. We would have one command for candidate functions and one for non-candidate functions. We would possibly need an auxiliary function to perform the candidate analysis.

Chapter 5

Related Work

In this chapter we will look at some of the theory behind the compilation scheme used in this thesis: monads and denotational semantics. Lastly, we will look at recent work on the extension of algebraic effects.

5.1 The Free Monad

Both the command tree and the `Compute` data type are a specialization of the free monad [26]. The free monad arises naturally when composing functors. In mathematical terms a functor is a mapping between categories [4]. In practical terms a functor is something that can be mapped over. A list is an example of a functor. We will take as our example functor a modified version of the `Maybe` data type and show what happens when we compose it with itself [12]. It describes a programming language where we either `Stop` with execution or sound a `Bell` and continue.

```
data Program a = Stop | Bell a

p0 :: Program (Program a)
p0 = Bell Stop

p1 :: Program (Program (Program a))
p1 = Bell (Bell Stop)
```

We see that our type grows with our expression. Both `p0` and `p1` should have the type `Program a`. We want a function of type `Program (Program a) -> Program a` that removes the nesting of functors. What we need is a fix-point of a functor.

```
data Fix f where
  Fix :: f (Fix f) -> Fix f
```

The type of `Fix` reflects the type of the function that we wanted. Our new programs will be of type `Fix Program`.

```
fp0 :: Fix Program
fp0 = Fix (Bell (Fix Stop))

fp1 :: Fix Program
fp1 = Fix (Bell (Fix (Bell (Fix Stop))))
```

`Fix` is almost a monad. What we need is a generic way to terminate programs. We will also need to parameterize over the return type instead of a functor. This leads us to the free monad.

```
data Free f a where
  Pure   :: a            -> Free f a
  Impure :: f (Free f a) -> Free f a
```

The free monad is thus a way to nest a functor, while maintaining a basic type of that functor. The free monad is restricted by this functor requirement. Without it, it is not a monad. Command trees do not have this requirement. We use a wrapper for our commands that ensures they are functors. This wrapper is a complex functor, but the principle is based on a simpler concept: pretending a mapping happened. The data type that captures this notion is the functor by construction `F`. It consists of something resembling a functor and a mapping over the contents of this fake functor. Whenever we map over `F`, we simply compose with the second argument. We pretend something happened; we update our mapping function.

```
data F f a where
  F :: f a -> (a -> b) -> F f b

instance Functor (F f) where
  fmap g (F f h) = F f (g . h)
```

If we extend this idea to the free monad we obtain the freer monad. The functor `Tps` uses is of a more specialized nature related to `Cps` and `Val`.

```
data G f a where
  G :: f -> [Val] -> (Val -> b) -> G f b

instance Functor (G f) where
  fmap g (G f ks k) = G f ks (g . k)
```

5.2 The Operational Monad

In this section we will derive the command tree monad and show how it relates to modular denotational semantics. We follow the style of the paper "Freer monads, more extensible effects" [15], which gives a derivation of the freer monad, a close relative of the command tree. The main problem the command tree and its relatives address is: expressing side-effectful computation in a composable/modular manner. We will start by unpacking this definition.

A side-effect can be understood as an interaction of an expression with its context [6]. A concrete example is the communication between a number of clients and a central server. A side-effect is a request from a client to the server. This can be a request for some data or an action. From this it follows immediately that we can model side effects with data types that specify such a request. A trivial example is a ping request:

```
data Request = Ping
```

The result of the ping request may be used in another part of the program. Our `Request` data type does not indicate what the return type of a ping request will be. Moreover, there is no place where the reply of the request is bound. We can define a data type that helps us with both these problems. We will call it `Compute`.

```
data Compute = Done Int | Compute Request (Int -> Compute)
```

`Compute` has two constructors that have the same role as those of the command tree. The `Done` constructor represents a computation without side effects that returns an integer. `Compute` binds `Requests` together. The result of a request is an integer and may be used in the following computations. An example shows the data types in action. We send two sequential ping requests to the central server and bind them to the variables `i` and `j`. Finally, we return the average of both. How a ping request is actually implemented is left open. `Request` just provides the interface for effectful computation.

```
pingtwiceavg = Compute Ping (\ i -> Compute Ping (\ j -> Done ((i + j) `div` 2)))
```

This way of defining sequences of effectful computations is called operational [1]. It is an alternative to other monadic implementations of side-effects. The upside of operational monads is their compositional nature. Traditional implementation of side-effects such as monad transformers [16] also compose, but suffer from non-commutative behavior [10].

Haskell provides typeclasses for defining monads. These are similar to an interface. Our `Compute` data type does not qualify for these, because Haskell requires monads to be parameterized over a type. We define our own `bind` function, which composes two computations; the `unit` function creates a trivial computation. In the implementation of the `bind` function we see that it pushes a function from an integer to a computation into the continuation of a computation. This is similar to list concatenation.

```
bind (Done i)      f = f i
bind (Compute r k) f = Compute r (\ i -> bind (k i) f)
-- unit is trivial
unit = Done
```

With our monadic definitions in place we can create a pretty version of our previous program that takes the average ping. We will define a helper function that represent a program that sends a ping request and returns the result. This enables us to `bind` these smaller programs together.

```
ping = Compute Ping Done

prettyping =
  ping `bind` \ i ->
  ping `bind` \ j ->
  unit ((i + j) `div` 2)
```

If we substitute our requests for commands and add subcontinuations to `Compute` we obtain our command tree.

In this and the previous section we have shown that command trees arise when we want to nest functors or model requests. This corresponds nicely to the objectives of modular denotational semantics where we work with semantic language modules that may have effects.

5.3 Modular Denotational Semantics

In this section we will give a short introduction to modular denotational semantics and then show how it relates to the work on the LamToWat compiler in this thesis. We will begin by stating some definitions. Denotational semantics is a method of giving meaning to programs by constructing mathematical objects which we often call 'values'. A programming language consists of multiple interacting parts called 'terms', like a function module or an arithmetic module. Terms can be given corresponding values separately. Then the meaning of the combined modules which make up a language is simply the sum or composition of these mappings. There are two popular approaches to solving this problem: algebraic effects [6] and monad transformers [16]. To give meaning to a program we write a program called an 'interpreter' that maps terms in each module to their respective denotation.

A denotational semantics is made up of three things: terms, values, and effects. We represent all three with Haskell constructs. A semantics is modular when we are able to split and extend all three components. In Haskell open unions give us the power to do this for terms and values. To model effects we will need monads: either monad transformers or the free monad depending on which approach we choose.

Now we will give an example of a modular denotational semantics for a simple language that can manipulate one memory cell `fetch, set` and do addition `add, int`. We will illustrate

the relation between monad transformers and algebraic effects [23]. One can be translated into the other and vice-versa. Both approaches allow one to construct an interpreter for our toy language. We will use Haskell typeclasses to represent modular interfaces for terms and highlight the relation by providing instances for both. The typeclasses are defined as follows:

```
class Cell d where
  fetch :: d
  set :: d -> d

class Addition d where
  add :: d -> d -> d
  int :: Int -> d
```

The domain of our language will simply be Haskell integers `Int`. We said before that our domain must also be extendible, we can also do that here by using open unions and describing integers as an element of the union like so: `Int :<: dom => ... dom ...`. However, this would lead to syntactic overhead which would only hinder the illustrative purposes of this example.

The monad transformer approach uses the state monad transformer to implement the effects of the `Cell` typeclass. It is included in the `mtl` package and is defined as follows, where `s` is the type of the internal state, `m` is a monad, and `a` is the return type.

```
newtype StateT s (m :: * -> *) a = StateT {runStateT :: s -> m (a, s)}
```

We can supply (modular) instances for both our typeclasses by adding a `MonadState` constraint, which is a typeclass in itself. This means that we get access to the operations `get, put`, which are very similar to `fetch, set`.

```
instance MonadState Int m => Cell (m Int) where
  fetch = get
  set d = do
    d' <- d
    put d'
    return d'

instance MonadState Int m => Addition (m Int) where
  add a b = do
    i <- a
    j <- b
    return (i + j)
  int = return
```

The Algebraic Effects approach requires a bit more preliminary work, because the monad is not taken from a package. We first define a new version of the operational/free monad that is specific to our domain of integers. We also define two command representing the `get, put` from the state monad transformer. We create smart constructors for both called `get'` and `put'`, respectively.

```
data Freer' cmd d where
  Pure :: d -> Freer' cmd d
  Impure :: cmd -> (Int -> Freer' cmd d) -> Freer' cmd d

data Cmd = Get | Put Int
```

We can now give instances for our typeclasses, which look almost the same. We now constrain our commands instead of an effect typeclass.

```
instance Cmd :<: cmd => Cell (Freer' cmd Int) where
  fetch = get'
  set d = do
    d' <- d
    put' d'

instance Cmd :<: cmd => Addition (Freer' cmd Int) where
  add a b = do
    i <- a
    j <- b
    return (i + j)
  int = return
```

We will also have to define a handler for `Cmd` commands called `hCmd` as follows.

```
hCmd :: Cmd :<: cmd => Freer' cmd a -> Int -> Freer' cmd (a, Int)
hCmd (Pure d)       s = Pure (d,s)
hCmd (Impure cmd k) s = case prj cmd of
  Just Get     -> hCmd (k s) s
  Just (Put s) -> hCmd (k s) s
  _            -> Impure cmd (\ d -> hCmd (k d) s)
```

We see that both approaches allow us to define instances for our language terms modularly. Both approaches use constraints to make the instances modular: monad transformers use typeclass constraints, while algebraic effects use constraints on the open union of commands.

Why did we use modular denotational semantics to write a compiler? Denotational semantics are very useful because they lead to a method of proving equality of terms of a language. When two terms have the same denotational value, they have the same meaning and one can be replaced by the other. This gives rise to many optimizations, which are provably correct.

One of the goals of a compiler is to optimize code, thus one can see where denotational semantics comes in. Moreover, it is important that the meaning of a program is not changed during compilation. In this thesis we have tried to extend this concept of denotational semantics from interpreters to compilers. In the second version of LamToWat we used algebraic effects represented by command trees.

5.4 Extending Algebraic Effects

Algebraic effects are able to model many effectful operations. However, there are some constructs that can not be given a proper semantics using algebraic effects. In this thesis we specialized our semantic command trees to syntactic command trees in order to perform variable analysis for closure conversion. It may be possible to instead generalize the notion of effects to tackle this problem. In this section we discuss two extensions of algebraic effects: scoped effects and latent effects.

The paper "Effect handlers in scope" [31] presents two approaches to extending algebraic effect with scoping constructs. Scoping constructs are constructs that are given meaning by scoping handlers, such as exception handlers. What makes scoping handlers different from normal handlers is that they create a local scope in which the impact of an effect is contained. The two roles of scoping handlers, semantics and scoping, conflict and lead to undesired interaction. By decoupling semantics and scoping and incorporating scoping in the syntax this problem may be solved. The first solution presented in the paper adds scope markers to the syntax of algebraic effects, which indicate the start and end of a scope. This solution

plays nicely with free monad infrastructure already present, but gives the user the possibility to go wrong by having unbalanced markers. The second solution uses higher-order syntax and requires a new version of the free monad. A specification of how a handler traverses higher-order signatures (commands) is needed too. The upside of higher-order syntax is their expressiveness and no risk of unbalanced markers.

The paper "Latent Effects" [7] builds on the idea of scoped effects. Latent effects may postpone computations for later execution. They address the requirements that algebraic effects should always be expressible as a function with a certain type and satisfy the algebraic property. An example of a postponed computation is a function abstraction, which can not be modeled with algebraic effects. The central idea of latent effects is that when a handler is applied to a signature it may decorate the return type of a computation with latent effects. Latent effects live inside latent effect trees, which are an even higher-order version of the free monads presented in the scoped effects paper.

Chapter 6

Conclusion

In this thesis we have examined how CPS can be improved upon as an IR in three ways: simpler control flow specification, typed transformations, and a modular interface. We have proposed command trees as a solution, which help both language implementers and compiler writers. Our new compilation scheme is validated by using it to implement LamToWat, a compiler that translates the lambda calculus into WebAssembly.

We started by discussing a reference implementation of LamToWat and showed how the source code is transformed by CPS conversion, closure conversion, and emitting. We identified three shortcomings of the IR: complex specification of CPS conversion, no types to show that functions are not nested, and constructor duplication when emitting.

We proposed command trees as the solution to these shortcomings. There are two types of command trees: semantic and syntactic. We had to split our data type because the semantic command trees did not allow for variable analysis, which is essential to closure conversion. We conjecture the possibility of an isomorphism between the two types of command trees.

The monadic nature of command trees allowed us to bind commands together and write the CPS conversion step easily. Open unions provided a modular approach to compilation by allowing the programmer to extend her set of commands. By having each transformation change specific command modules we clearly indicated what it altered. The implementation of command trees in this thesis can be extended in many aspects. We proposed some improvements of command trees, such as type constraints to show the absence of free variables or open unions that behave more like real sets.

We tested both implementations of LamToWat by compiling and running programs in the lambda calculus and checking their results. We listed a number of improvements with respect to the compiler itself: optimized closures and the inlining of functions.

Bibliography

[1] Heinrich Apfelmus. The Operational Monad Tutorial. https://apfelmus.nfshost.com/articles/operational-monad.html. Accessed: 2020-12-08.

[2] Andrew W. Appel. Compiling with Continuations (corr. version). Cambridge University Press, 2006. ISBN: 978-0-521-03311-4.

[3] Hendrik P Barendregt et al. The lambda calculus. Vol. 3. North-Holland Amsterdam, 1984.

[4] Michael Barr and Charles Wells. Category theory for computing science. Vol. 49. Prentice Hall New York, 1990.

[5] Chiel Bruin. "Dynamix on the Frame VM: Declarative dynamic semantics on a VM using scopes as frames". In: (2020). URL: http://resolver.tudelft.nl/uuid:ddedce14-65ad-4f16-912e-6b0658eaecc0.

[6] Robert Cartwright and Matthias Felleisen. "Extensible Denotational Language Specifications". In: Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings. Ed. by Masami Hagiya and John C. Mitchell. Vol. 789. Lecture Notes in Computer Science. Springer, 1994, pp. 244–272. ISBN: 3-540-57887-0. DOI: 10.1007/3-540-57887-0_99. URL: https://doi.org/10.1007/3-540-57887-0%5C_99.

[7] Anonymous Casper Bach Poulsen. "Latent Effects". In: (2021).

[8] Adam Chlipala. "A certified type-preserving compiler from lambda calculus to assembly language". In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. Ed. by Jeanne Ferrante and Kathryn S. McKinley. ACM, 2007, pp. 54–65. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250742. URL: https://doi.org/10.1145/1250734.1250742.

[9] Alonzo Church. The Calculi of Lambda Conversion.(AM-6), Volume 6. Princeton University Press, 2016.

[10] Laurence E. Day and Graham Hutton. "Compilation à la Carte". In: Proceedings of the 25th Symposium on Implementation and Application of Functional Languages, Nijmegen, The Netherlands, August 28-30, 2013. Ed. by Rinus Plasmeijer. ACM, 2013, p. 13. ISBN: 978-1-4503-2988-0. DOI: 10.1145/2620678.2620680. URL: https://doi.org/10.1145/2620678.2620680.

[11] extensible-effects | Hackage. https://hackage.haskell.org/package/extensible-effects. Accessed: 2021-01-04.

[12] Gabriel Gonzalez. Why free monads matter. http://www.haskellforall.com/2012/06/you-could-have-invented-free-monads.html. Accessed: 2020-12-09.

[13] Louis-Julien Guillemette and Stefan Monnier. "A type-preserving closure conversion in haskell". In: Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007. Ed. by Gabriele Keller. ACM, 2007, pp. 83–92. ISBN: 978-1-59593-674-5. DOI: 10.1145/1291201.1291212. URL: https://doi.org/10.1145/1291201.1291212.

[14] Haskell Home Page. https://haskell.org. Accessed: 2020-12-01.

[15] Oleg Kiselyov and Hiromi Ishii. "Freer monads, more extensible effects". In: Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015. Ed. by Ben Lippmeier. ACM, 2015, pp. 94–105. ISBN: 978-1-4503-3808-0. DOI: 10.1145/2804302.2804319. URL: https://doi.org/10.1145/2804302.2804319.

[16] Sheng Liang, Paul Hudak, and Mark P. Jones. "Monad Transformers and Modular Interpreters". In: Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 333–343. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199528. URL: https://doi.org/10.1145/199448.199528.

[17] Sam Lindley and Conor McBride. "Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming". In: SIGPLAN Not. 48.12 (Sept. 2013), pp. 81–92. ISSN: 0362-1340. DOI: 10.1145/2578854.2503786. URL: https://doi.org/10.1145/2578854.2503786.

[18] J. Gregory Morrisett et al. "From System F to Typed Assembly Language". In: POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998. Ed. by David B. MacQueen and Luca Cardelli. ACM, 1998, pp. 85–97. ISBN: 0-89791-979-3. DOI: 10.1145/268946.268954. URL: https://doi.org/10.1145/268946.268954.

[19] open-union | Hackage. https://hackage.haskell.org/package/open-union. Accessed: 2021-01-04.

[20] parsec: Monadic parser combinators. https://hackage.haskell.org/package/parsec/. Accessed: 2021-04-29.

[21] Gordon D. Plotkin and Matija Pretnar. "Handling Algebraic Effects". In: Log. Methods Comput. Sci. 9.4 (2013). DOI: 10.2168/LMCS-9(4:23)2013. URL: https://doi.org/10.2168/LMCS-9(4:23)2013.

[22] Casper Bach Poulsen. Compilers for Free. http://casperbp.net/posts/2020-04-compilers-for-free/draft/#free-monad. Accessed: 2020-12-06.

[23] Tom Schrijvers et al. "Monad transformers and modular algebraic effects: what binds them together". In: Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019. Ed. by Richard A. Eisenberg. ACM, 2019, pp. 98–113. ISBN: 978-1-4503-6813-1. DOI: 10.1145/3331545.3342595. URL: https://doi.org/10.1145/3331545.3342595.

[24] Shake Build System. https://shakebuild.com/. Accessed: 2021-01-25.

[25] Guy L Steele Jr. "Rabbit: A compiler for Scheme". In: (1978).

[26] Wouter Swierstra. "Data types à la carte". In: Journal of functional programming 18.4 (2008), p. 423.

[27] The Haskell Cabal | Overview. https://www.haskell.org/cabal/. Accessed: 2021-03-31.

[28] WABT: The WebAssembly Binary Toolkit. https://github.com/webassembly/wabt. Accessed: 2021-01-17.

[29] WebAssembly Home Page. https://webassembly.org. Accessed: 2020-12-01.

[30] Nicolas Wu and Tom Schrijvers. "Fusion for Free - Efficient Algebraic Effect Handlers". In: Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings. Ed. by Ralf Hinze and Janis Voigtländer. Vol. 9129. Lecture Notes in Computer Science. Springer, 2015, pp. 302–322. ISBN: 978-3-319-19796-8. DOI: 10.1007/978-3-319-19797-5_15. URL: https://doi.org/10.1007/978-3-319-19797-5%5C_15.

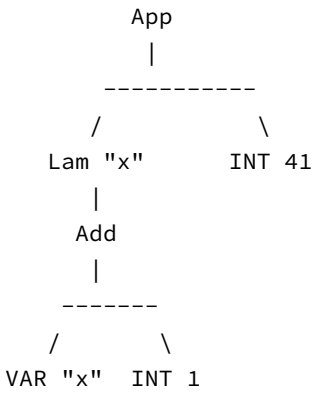
[31] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. "Effect handlers in scope". In: Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014. Ed. by Wouter Swierstra. ACM, 2014, pp. 1–12. ISBN: 978-1-4503-3041-1. DOI: 10.1145/2633357.2633358. URL: https://doi.org/10.1145/2633357.2633358.

Appendix A

A

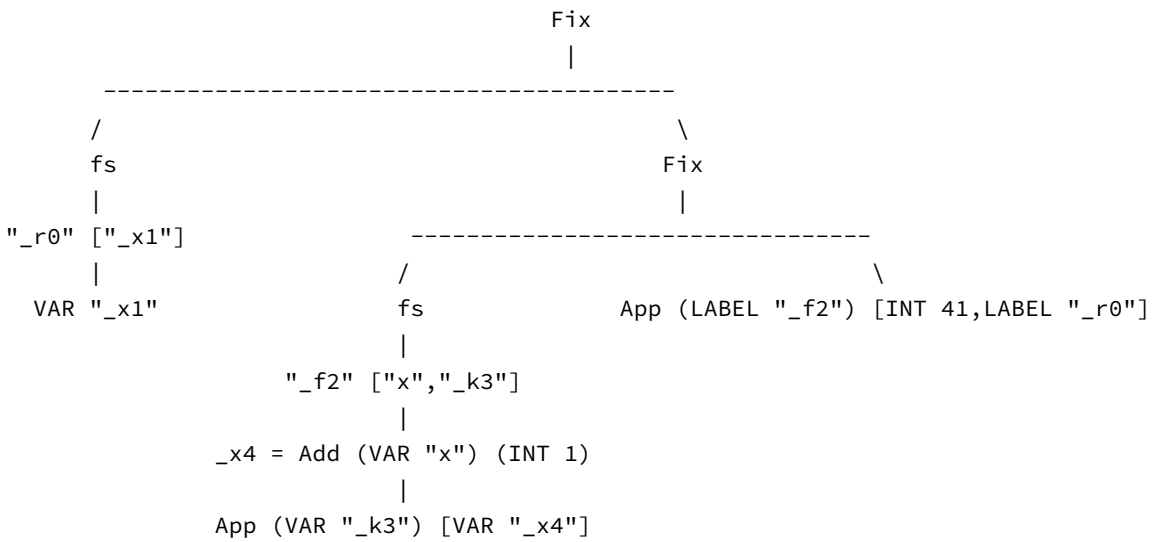
A.1 LamToWat Version 1 Transformation of: ((\x -> x + 1) 41)

$$String \xrightarrow{parse} \boxed{Lam} \xrightarrow{lam2cps} Cps \xrightarrow{cps2cps} Cps \xrightarrow{cps2wat} Wat \xrightarrow{emit} String$$



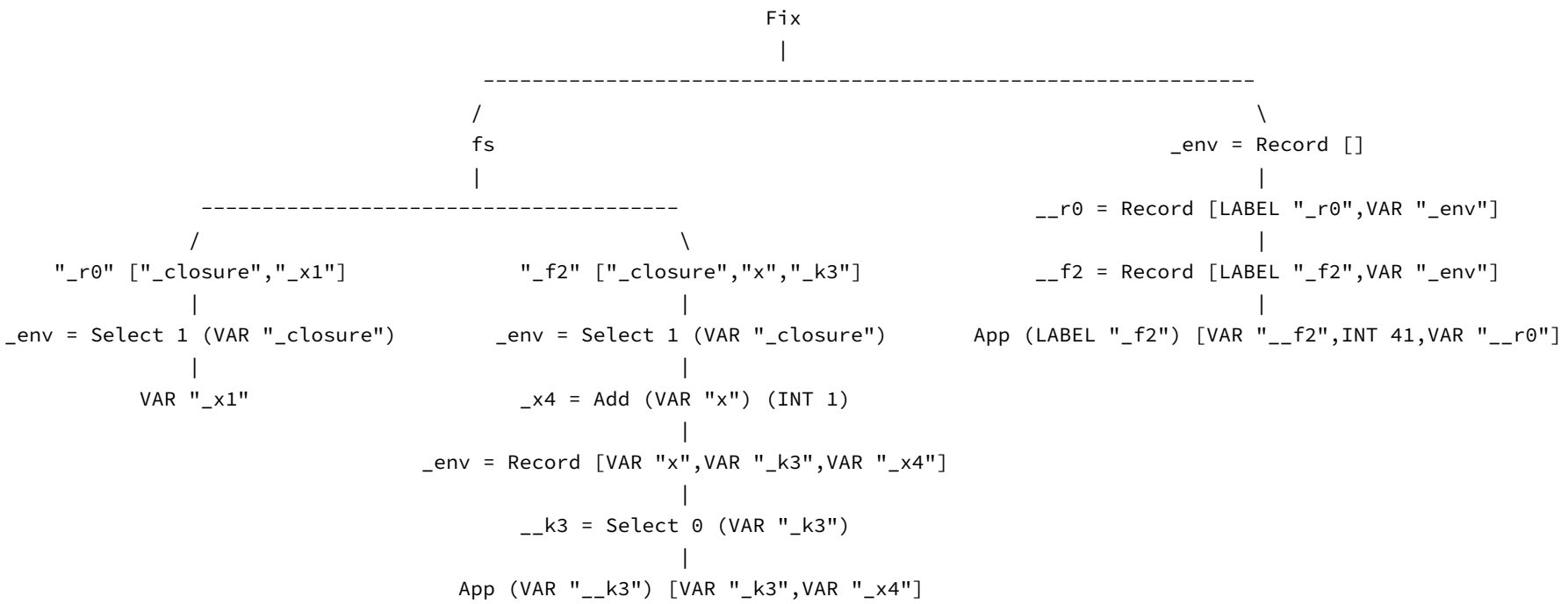
Graph representation of the AST of our simple lambda expression `(\ x -> x + 1) 41`.

$$String \xrightarrow{parse} Lam \xrightarrow{lam2cps} \boxed{Cps} \xrightarrow{cps2cps} Cps \xrightarrow{cps2wat} Wat \xrightarrow{emit} String$$



In the graph above we see how our simple lambda calculus program is converted with `lam2cps`. Freshly generated variables are prefixed with a `_`. We see two functions `_f2` and `_r0` have been created. The first represents our original function with an additional continuation argument `_k3`. The body of the function adds the number one to `x` and passes the result to the continuation. The `_r0` function represents a return point. In this case the return point of the entire program as it uses the `Val` constructor in its body. Finally, we see the call to `_f2` where we pass the original argument and the `_r0` function as the continuation.

$$String \xrightarrow{parse} Lam \xrightarrow{lam2cps} Cps \xrightarrow{cps2cps} \boxed{Cps} \xrightarrow{cps2wat} Wat \xrightarrow{emit} String$$



$$String \xrightarrow{parse} Lam \xrightarrow{lam2cps} Cps \xrightarrow{cps2cps} Cps \xrightarrow{cps2wat} \boxed{Wat} \xrightarrow{emit} String$$

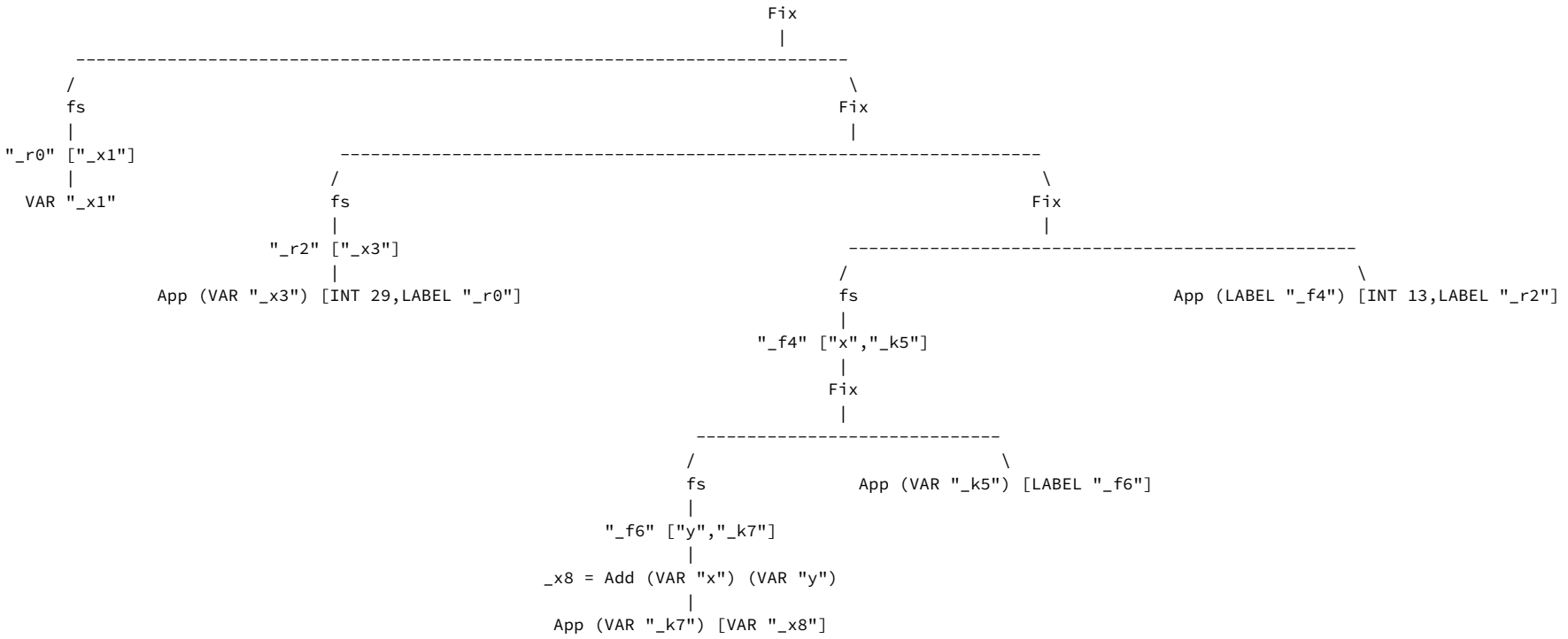
```
                                    Fix
                                     |
                  ---------------------------------------------------------
                 /                                                         \
                 fs                                                    _env = Malloc 0
                 |                                                         |
          ----------------------------------                         __r0 = Malloc 2
         /                                  \                            |
    "_r0" ["_closure","_x1"]          "_f2" ["_closure","x","_k3"]     Store 0 (VAR "__r0") (INT 0)
         |                                  |                            |
_env = Load 1 (VAR "_closure")      _env = Load 1 (VAR "_closure")   Store 1 (VAR "__r0") (VAR "_env")
         |                                  |                            |
      VAR "_x1"                       _x4 = Add (VAR "x") (INT 1)     __f2 = Malloc 2
                                            |                            |
                                      _env = Malloc 3                 Store 0 (VAR "__f2") (INT 1)
                                            |                            |
                                   Store 0 (VAR "_env") (VAR "x")    Store 1 (VAR "__f2") (VAR "_env")
                                            |                            |
                                   Store 1 (VAR "_env") (VAR "_k3")  App (INT 1) [VAR "__f2",INT 41,VAR "__r0"]
                                            |
                                   Store 2 (VAR "_env") (VAR "_x4")
                                            |
                                      __k3 = Load 0 (VAR "_k3")
                                            |
                                App (VAR "__k3") [VAR "_k3",VAR "_x4"]
```

A.2 LamToWat Version 1 Transformation of: ((\x y -> x + y) 13 29)

$$String \xrightarrow{parse} \boxed{Lam} \xrightarrow{lam2cps} Cps \xrightarrow{cps2cps} Cps \xrightarrow{cps2wat} Wat \xrightarrow{emit} String$$

Graph representation of the AST of our simple lambda expression `(\ x y -> x + y) 13 29`.

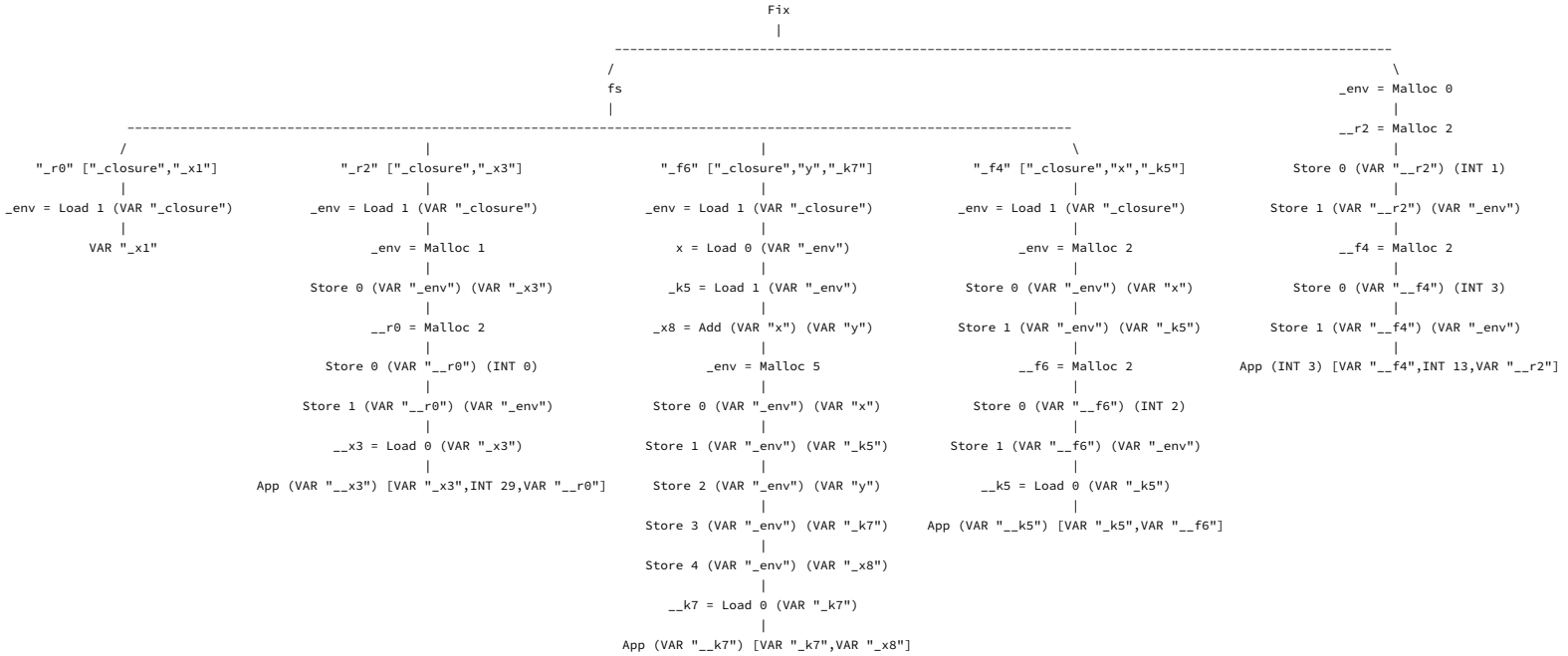
$$String \xrightarrow{parse} Lam \xrightarrow{lam2cps} \boxed{Cps} \xrightarrow{cps2cps} Cps \xrightarrow{cps2wat} Wat \xrightarrow{emit} String$$



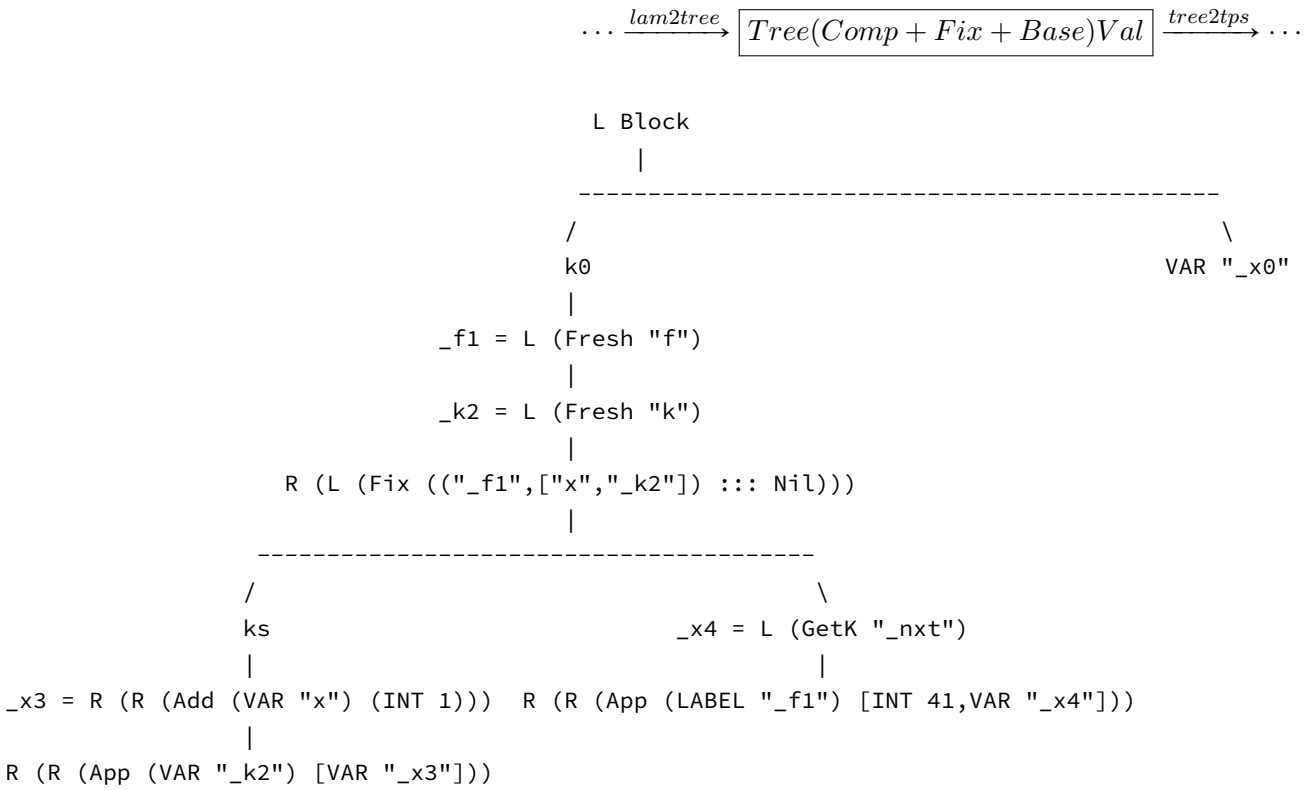
$$String \xrightarrow{parse} Lam \xrightarrow{lam2cps} Cps \xrightarrow{cps2cps} \boxed{Cps} \xrightarrow{cps2wat} Wat \xrightarrow{emit} String$$

If we take a look at the above graphic, the first thing we notice is that functions are no longer nested; there is only one, top-level `FIX`. We also see the addition of `Record` and `Select` expressions. We have four functions representing our original two functions and the two applications. We can see how the original function bodies are prefixed with opening closures and postfixed with creating closures before application. All functions have an extra `_closure` argument. If a function was originally nested more deeply, it will open and create a closure with more elements, because there will be more variables in scope. This is the case for the function `_f6`, where the addition of `x` and `y` happens. The first closure that is created in our program before calling `_f4` is empty. This is because at the top level there are no free variables. The two closures for `_f4` and `_r2` named `__f4` and `__r2`, respectively, are a sort of dummy closure only containing a function pointer and a empty environment.

$$String \xrightarrow{parse} Lam \xrightarrow{lam2cps} Cps \xrightarrow{cps2cps} Cps \xrightarrow{cps2wat} \boxed{Wat} \xrightarrow{emit} String$$



A.3 LamToWat Version 2 Transformation of: ((\x -> x + 1) 41)



The most notable novelty in the command tree based CPS conversion is the use of the extensible sum constructors `L` and `R` and the use of a `Block`. `Block`s are used to bind parts of code together and replace the metacontinuation of the original version. If we take the code outside the block it would end with an application of `_f1`. Since applications do not have a continuation, any code after it would be discarded. Since in this case the application is the last thing that happens it does not really matter, but when we more than one application in our expression it becomes a problem. Continuations will be dropped. This is not the behavior we want, so we use `Block`s instead.

$$\cdots \xrightarrow{tree2tps} \boxed{Tps(Fix + Base)Val} \xrightarrow{hClos} \cdots$$

```
                              L (Fix (("_r0",["_x1"]) ::: Nil))
                                                      |
     ------------------------------------------------
    /                                                  \
    ks                               L (Fix (("_f2",["x","_k3"]) ::: Nil))
    |                                                  |
VAR "_x1"                     ----------------------------------------
                             /                                        \
                             ks                   R (L (App (LABEL "_f2") [INT 41,LABEL "_r0"]))
                             |
           _x4 = R (L (Add (VAR "x") (INT 1)))
                             |
           R (L (App (VAR "_k3") [VAR "_x4"]))
```

After `tree2tps` our example no longer contains blocks. They have been replaced by continuation functions. In our example this means that block `_b0` has been replaced by function `_r0`. We see that no more `fresh` commands are present. The program is the same as the program of the original version before closure conversion up to renaming of variables. The structure of `Fix` is also a bit different, as the function names and parameters are stored inside the `Fix` command and the function bodies are stored in the subcontinuations `ks`. We have handled the extra compilation commands that were used to describe control flow `Block` and generate fresh variable names `Fresh`.

$$\cdots \xrightarrow{hClos} \boxed{Tps(Fix + Record + Base)Val} \xrightarrow{hRecord} \cdots$$

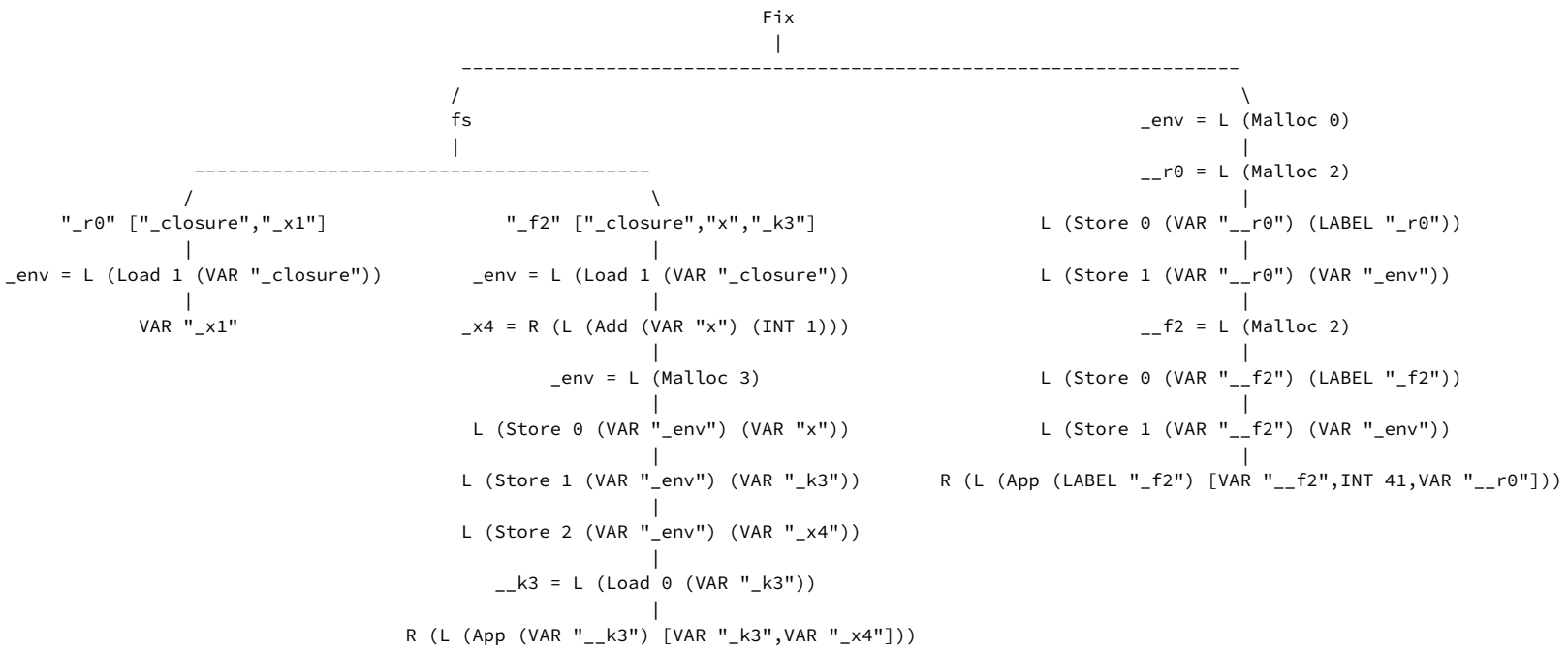
```
                                                    R (L (Fix (("_r0",["_closure","_x1"]) ::: Nil)))
                                                                        |
                  -----------------------------------------------------------------------------
                  /                                                                             \
                  ks                                                 R (L (Fix (("_f2",["_closure","x","_k3"]) ::: Nil)))
                  |                                                                             |
_env = L (Select 1 (VAR "_closure"))                            ---------------------------------------------------------
                  |                                             /                                                        \
             VAR "_x1"                                          ks                                            _env = L (Record [])
                                                                |                                                        |
                                              _env = L (Select 1 (VAR "_closure"))                 __r0 = L (Record [LABEL "_r0",VAR "_env"])
                                                                |                                                        |
                                            _x4 = R (R (L (Add (VAR "x") (INT 1))))                __f2 = L (Record [LABEL "_f2",VAR "_env"])
                                                                |                                                        |
                                        _env = L (Record [VAR "x",VAR "_k3",VAR "_x4"])    R (R (L (App (LABEL "_f2") [VAR "__f2",INT 41,VAR "__r0"])))
                                                                |
                                                __k3 = L (Select 0 (VAR "_k3"))
                                                                |
                                       R (R (L (App (VAR "__k3") [VAR "_k3",VAR "_x4"])))
```

After closure conversion functions have an extra `_closure` argument. Bodies of functions are prefixed with opening the closure and postfixed with creating closures before calling another function. In the previous version of LamToWat we did not get to see this rendering of our example, because it was hoisted immediately.

$$\cdots \xrightarrow{hRecord} \boxed{Tps(Fix + Malloc + Base)Val} \xrightarrow{hFix} \cdots$$

```
                                                R (L (Fix (("_r0",["_closure","_x1"]) ::: Nil)))
                                                                        |
                 ---------------------------------------------------------------------------------
                /                                                                                 \
               ks                                                                                  R (L (Fix (("_f2",["_closure","x","_k3"]) ::: Nil)))
               |                                                                                                          |
_env = L (Load 1 (VAR "_closure"))                                                  ---------------------------------------------------------
               |                                                                   /                                                          \
          VAR "_x1"                                                               ks                                                      _env = L (Malloc 0)
                                                                                  |                                                          |
                                                                     _env = L (Load 1 (VAR "_closure"))                                  __r0 = L (Malloc 2)
                                                                                  |                                                          |
                                                                  _x4 = R (R (L (Add (VAR "x") (INT 1))))                    L (Store 0 (VAR "__r0") (LABEL "_r0"))
                                                                                  |                                                          |
                                                                           _env = L (Malloc 3)                               L (Store 1 (VAR "__r0") (VAR "_env"))
                                                                                  |                                                          |
                                                                     L (Store 0 (VAR "_env") (VAR "x"))                                  __f2 = L (Malloc 2)
                                                                                  |                                                          |
                                                                   L (Store 1 (VAR "_env") (VAR "_k3"))                  L (Store 0 (VAR "__f2") (LABEL "_f2"))
                                                                                  |                                                          |
                                                                   L (Store 2 (VAR "_env") (VAR "_x4"))                  L (Store 1 (VAR "__f2") (VAR "_env"))
                                                                                  |                                                          |
                                                                     __k3 = L (Load 0 (VAR "_k3"))           R (R (L (App (LABEL "_f2") [VAR "__f2",INT 41,VAR "__r0"])))
                                                                                  |
                                                                R (R (L (App (VAR "__k3") [VAR "_k3",VAR "_x4"])))
```

$$\cdots \xrightarrow{hFix} \boxed{(Fix(Tps(Malloc + Base)Val))} \xrightarrow{tps2wat} \cdots$$

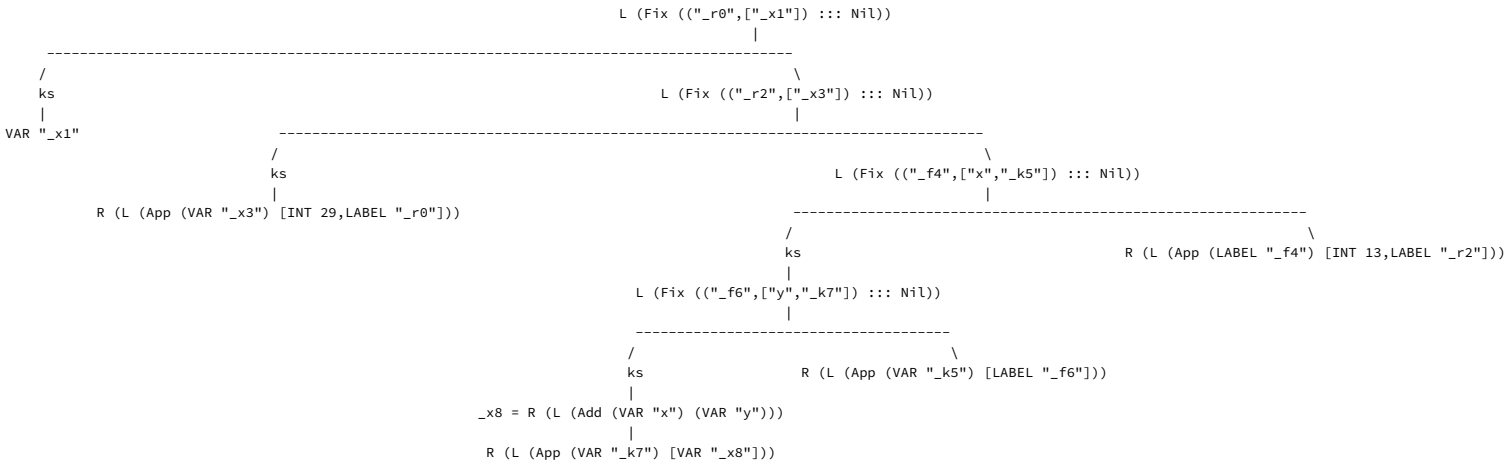


$$\ldots \xrightarrow{tps2wat} \boxed{Wat} \xrightarrow{emit} String$$

A.4 LamToWat Version 2 Transformation of: ((\x y -> x + y) 13 29)

$$\cdots \xrightarrow{lam2tree} \boxed{Tree(Comp + Fix + Base)Val} \xrightarrow{tree2tps} \cdots$$

$$\cdots \xrightarrow{tree2tps} \boxed{Tps(Fix + Base)Val} \xrightarrow{hClos} \cdots$$



$$\cdots \xrightarrow{hClos} \boxed{Tps(Fix + Record + Base)Val} \xrightarrow{hRecord} \cdots$$

$$\cdots \xrightarrow{hRecord} \boxed{Tps(Fix + Malloc + Base)Val} \xrightarrow{hFix} \cdots$$

$$\cdots \xrightarrow{hFix} \boxed{(Fix(Tps(Malloc + Base)Val))} \xrightarrow{tps2wat} \cdots$$

$$\ldots \xrightarrow{tps2wat} \boxed{Wat} \xrightarrow{emit} String$$