UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE

Master's thesis

Anders Kiel Hovgaard

Higher-order functions for a high-performance programming language for GPUs

Supervisors: Martin Elsman and Troels Henriksen

May, 2018

Abstract

General-purpose massively parallel processors, such as modern GPUs, have become common, but the difficulty of programming these machines is well known. Pure functional programming provides some reassurance that the situation can be improved, by guaranteeing referential transparency and providing useful combinators for expressing data-parallel computations. Unfortunately, one of the main features of functional programming, namely higher-order functions, cannot be efficiently implemented on GPUs by the usual means. In this thesis, we present a defunctionalization transformation that relies on type-based restrictions on the use of functions to be able to completely eliminate higher-order functions in all cases, without introducing any branching. We prove the correctness of the transformation and discuss its implementation in Futhark, a data-parallel functional language that generates GPU code. The use of these restricted higher-order functions has no impact on runtime performance and we argue that we gain many of the benefits of general higher-order functions without being hindered by the restrictions in most cases in practice.

Contents

Contents **ii**

1 Introduction **1**
1.1 Contributions . . . 2
1.2 Report outline . . . 4
1.3 Notation . . . 4

2 Background **5**
2.1 Reynolds's defunctionalization . . . 5
2.2 Graphics processing units . . . 6
2.3 Data parallelism and the Futhark language . . . 7

3 Language **8**
3.1 Syntax . . . 8
3.2 Type system . . . 9
3.3 Operational semantics . . . 11

4 Defunctionalization **14**

5 Metatheory **17**
5.1 Type soundness and normalization . . . 17
5.2 Translation termination and preservation of typing . . . 19
5.3 Preservation of meaning . . . 24
5.4 Correctness of defunctionalization . . . 30

6 Implementation **31**
6.1 Overview . . . 31
6.2 Polymorphism and defunctionalization . . . 32
6.3 Optimizations . . . 32
6.4 Array shape parameters . . . 34
6.5 In-place updates . . . 35

7 Empirical evaluation **37**
7.1 Performance evaluation . . . 37
7.2 Programming with restricted higher-order functions . . . 38

8 Extensions **46**
8.1 Support for function-type conditionals . . . 46
8.2 Support for while-loops and recursion . . . 48

9 Conclusion **51**
9.1 Related work 51
9.2 Future work 53
9.3 Closing remarks 54

Bibliography **55**

Chapter 1

Introduction

Higher-order functions are ubiquitous in functional programming. They enable programmers to write abstract and composable code, which enables the development of modular programs [20]. Functional languages are often considered well-suited for parallel programming, because of the presence of referential transparency and the lack of shared state and side effects, which helps prevent issues such as race conditions. Data-parallel programming, in particular, arises from the inherently parallel nature of many of the typical operations from functional languages, including map, reduce, and scan.

Modern hardware is increasingly parallel. While the previous exponential growth in single-thread performance of computing hardware has been flattening over the last decade, the number of individual processing units continues to grow. This development has transformed the subject of parallel programming from a research topic focusing on super-computers to a practical necessity to enable the efficient use of modern hardware.

The emergence of graphics processing units (GPUs) that allow for general-purpose programming has exacerbated the need for developing practical techniques for programming parallel hardware. In recent years, the theoretical peak performance of GPUs has grown to be much higher than that of CPUs. This disparity is due to the fundamentally different design philosophy of the massively parallel processors that GPUs have evolved into, driven by the demand for high-definition 3D computer graphics. However, programming GPUs to harness efficiently all the potential parallel performance for general-purpose computations is notoriously difficult, since GPUs offer a significantly more restricted programming model than that of CPUs.

We would like for the benefits of higher-order functions to be made available for functional GPU programming. Unfortunately, GPUs do not readily allow for higher-order functions to be implemented because they have only limited support for function pointers.

If higher-order functions cannot be implemented directly, we may instead opt to remove them by means of some program transformation that replaces them by a simpler language mechanism that is easier to implement. The canonical such transformation is *defunctionalization*, which was first described by Reynolds [26, 27] in the context of so-called *definitional interpreters*, that is, interpreters that mainly serve to define a language. Reynolds's defunctionalization represents each functional value by a uniquely tagged data value and

each application is replaced by a call to an *apply function*, which performs a case dispatch over all the possible tags and essentially serves as an interpreter for the functional values in the original program.

One of the major problems with this kind of transformation in the context of generating code for GPUs is that it introduces a large amount of branching into the transformed program. The most basic version of defunctionalization will add a case to the apply function for every function abstraction in the source program. This amount of branching is very problematic for GPUs because of the issue of *branch divergence*. Since neighboring threads in a GPU execute together in lockstep, a large amount of branching will cause many threads to be idle in the branches where they are not executing instructions.

Ideally, we want to eliminate higher-order functions without introducing an excessive amount of branching into the program. Clearly, we cannot in general determine the form of the applied function at every application site in a program, since this may depend on dynamic information that is only available at run-time. Consider, for example, a conditional of function type that depends on a dynamic condition or an array of functions indexed by a dynamic value.

By restricting the use of functions in programs, we are able to statically determine the form of the applied function at every application. Specifically, we disallow conditionals and loops from returning functional values, and we disallow arrays from containing functions. By enforcing these restrictions, we can translate a program using first-class and higher-order functions into a completely first-order program by specializing each application to the particular form of function that may occur at run-time. The result is essentially equivalent to completely inlining the apply function in a program produced by Reynolds-style defunctionalization. Notably, the transformation does not introduce any more branching than was already present in the original program.

We have used the *Futhark* [18] language to demonstrate this idea. Futhark is a data-parallel, purely functional array language with the main goal of generating high-performance parallel code. Although the language itself is hardware-agnostic, the main focus is on the implementation of an aggressively optimizing compiler that generates efficient GPU code via OpenCL.

To illustrate the basic idea of our defunctionalization transformation, we show a simple Futhark program in Figure 1.1a and the resulting program after defunctionalization in Figure 1.1b (simplified slightly for the sake of presentation). The result is a first-order program which explicitly passes around the closure environments, in the form of records capturing the free variables, in place of the first-class functions in the source program.

1.1 Contributions

The principal contributions of this thesis are the following:

- A defunctionalization transformation expressed on a simple data-parallel functional array language, with a type system that moderately restricts the use of first-class functions to allow for defunctionalization to effectively remove higher-order functions in all cases, without introducing any branching.

```
let twice (g: i32 -> i32) = \x -> g (g x)

let main =
  let f = let a = 5
          in twice (\y -> y+a)
  in f 1 + f 2
```

(a) Source program.

```
let g' (env: {a: i32}) (y: i32) =
  let a = env.a
  in y+a

let f' (env: {g: {a: i32}}) (x: i32) =
  let g = env.g
  in g' g (g' g x)

let main =
  let f = let a = 5
          in {g = {a = a}}
  in f' f 1 + f' f 2
```

(b) Target program.

Figure 1.1: Example demonstrating the defunctionalization transformation.

- A correctness proof of the transformation: a well-typed program will translate to another well-typed program and the translated program will evaluate to a value, corresponding to the value of the original program, or fail with an error if the original program fails.
- An implementation of defunctionalization in the compiler for the high-performance functional language Futhark, and a description of various extensions and optimizations.
- An evaluation of the implementation with respect to the performance of the compiled programs and the usefulness of the restricted higher-order functions.
- A description of how the restrictions on functions can be loosened to allow more programs to be defunctionalized, while only introducing minimal branching.

The work presented in this thesis has been condensed and submitted as an article to the symposium on Trends in Functional Programming (TFP) and will be presented at the event in Gothenburg in June 2018, in the form:

> Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. High-performance defunctionalization in Futhark. *Trends in Functional Programming*, 2018.

1.2 Report outline

The remainder of this thesis is organized as follows. In the next chapter, we briefly cover some preliminary topics. In Chapter 3, we define the language on which the defunctionalization transformation will operate, including the type system restrictions that are essential to the effectiveness of the transformation. In Chapter 4, we define the defunctionalization transformation itself. In Chapter 5, we present the metatheory of the system, namely proofs of type soundness, termination and typing preservation of defunctionalization, and meaning preservation of defunctionalization. In Chapter 6, we discuss the implementation in the Futhark compiler and the various extensions and optimizations that were made. In Chapter 7, we evaluate our implemented defunctionalization technique in practice, in terms of run-time performance and the usefulness of our restricted higher-order functions. In Chapter 8, we discuss how we could loosen the type restrictions and, in particular, allow function-type conditionals. Finally, in Chapter 9, we discuss related work, mention a few ideas for future work, and conclude.

1.3 Notation

We use the metanotation $(\mathcal{Z}_i)^{i \in 1..n}$ to denote a sequence of objects $\mathcal{Z}_1, \ldots, \mathcal{Z}_n$, where each $\mathcal{Z}_i$ may be a syntactic object, a derivation of a judgment, etc.

We may sometimes write $\mathcal{D} :: \mathcal{J}$ to give the name $\mathcal{D}$ to the derivation of the judgment $\mathcal{J}$ so that we can refer to it later.

Chapter 2

Background

In this chapter, we briefly cover a number of preliminary topics that will serve as background and help set the context for the remainder of the report. More background and related work will be discussed in Section 9.1.

2.1 Reynolds's defunctionalization

As mentioned earlier, the basic idea of defunctionalization was first described by John Reynolds [26]. In his seminal paper, Reynolds classified higher-order languages, defined via definitional interpreters, according to whether the defining language uses higher-order functions and whether the evaluation order of the defined language depends upon the evaluation order of the defining language. As part of this investigation, Reynolds informally described a defunctionalization method for converting a higher-order interpreter to a first-order equivalent, by representing each λ-abstraction occurring in the original program by a tagged record that captures the free variables in the function and replacing each application by a call to an apply function that interprets the record expressions representing the original functions.

Defunctionalization is similar to closure conversion in that it makes the environments explicit in programs, but unlike closure conversion which represents the code part of closures by a pointer to the code, defunctionalization simply stores a tag that uniquely identifies a function. Each application is then translated to a case dispatch on these tags rather than an indirect jump.

We now show a simple example to illustrate this kind of defunctionalization transformation. Consider the following Haskell program:

```
main = let a = 1
           f = \x -> x+a
           g = \x -> \y -> f x + y
       in g 3 5
```

We encode the λ-abstractions in the program as data using an algebraic data type `F`, with one value constructor for each abstraction, with the free variables of that abstraction attached to it. For instance, we represent the abstraction `(\y -> f x + y)` with the data value `F3 f x`, where `f` has itself been encoded.

```
data F = F1 Int | F2 F | F3 F Int
```

The program is then translated as follows:

```
apply h z =
  case h of F1 a   -> z+a
            F2 f   -> F3 f z
            F3 f x -> apply f x + z

main = let a = 1
           f = F1 a
           g = F2 f
       in apply (apply g 3) 5
```

Notice that the `apply` function is actually ill-typed, although the program is still safe and semantically equivalent to the original program. The same is true for the apply function in the original presentation by Reynolds and this issue has been treated in multiple later works. We will mention some of those in our discussion of related work in Section 9.1.

2.2 Graphics processing units

Modern graphics processing units (GPUs) are massively parallel processors with a programming model that differs significantly from that of traditional sequential and even multi-core CPUs. Mainstream CPUs are characterized by high sequential performance, flexible control flow, and low-latency memory access through the use of a large cache hierarchy. On the other hand, GPUs focus on high total throughput, dedicating more transistors to processing units and less to caches and control logic, while sacrificing sequential performance and low memory access latency. GPUs support very high bandwidth memory access, assuming specific memory access patterns (coalesced memory access).

A GPU consists of a number of *multiprocessors*, each of which consists of a number of *streaming processors*. Each streaming processor has a number of hardware-supported threads, which execute together in so called *warps* of usually 32 threads, which is the basic unit of scheduling in streaming processors. The threads in a warp execute in a single instruction multiple data (SIMD) fashion, that is, each thread executes the same instruction at any given time in lockstep on different parts of the data. If some threads of a warp take different branches in a conditional, each branch is executed one after the other with some threads being inactive in one branch and the other threads being inactive in another branch. This kind of *branch divergence* can cause much inefficiency if many warps suffer from this problem.

A large number of threads is usually needed to expose a sufficient amount of parallelism on a GPU. This significantly limits the amount of memory that is available to each thread. In particular, threads do not have a stack as we know from CPUs. While stacks can be emulated at some cost in efficiency, they can only have a very limited size because of this.

Because of the lockstep execution of neighboring threads and because of the limited amount of memory available to each thread, GPUs do not offer efficient support (if any) for function pointers and recursion. Thus, the usual approach to implementing higher-order functions, by some method based on closure conversion and function pointers, is not feasible on GPUs. While Reynolds-style defunctionalization could circumvent the issue with lack of proper support for

function pointers, the transformation introduces too much branching into the program, which would most likely be very inefficient when executed on a GPU.

This work is primarily concerned with the implementation of higher-order functions on GPUs, although other accelerator devices exist.

2.3 Data parallelism and the Futhark language

Data parallelism is a model of parallel execution where multiple threads execute the same operation on different parts of the data. This is in contrast to *task parallelism* where multiple threads execute different instructions, independently, on different pieces of data. Data-parallel programming maps well to execution on GPUs because of their massively parallel architecture and the SIMD style lockstep execution of warps, as mentioned before. This thesis is exclusively concerned with data-parallel programming.

Futhark [18, 17] is a data-parallel, purely functional array language. In Futhark, parallelism is expressed through the use of *second-order array combinators* (SOACs), such as `map`, `reduce`, `scan`, and `filter`, which resemble the higher-order functions found in conventional functional languages. These SOACs have sequential semantics, but the Futhark compiler can exploit them for generating parallel code. Futhark also supports nested parallelism. As mentioned, GPUs do not properly support stacks, so for this reason Futhark does not support recursion. Instead, Futhark offers a few different sequential looping constructs, corresponding to certain tail-recursive functions.

Before the work of this thesis, Futhark was a first-order language, only allowing partially applied functions and λ-abstractions as arguments to the built-in SOACs and no other places.

Futhark makes use of a higher-order module system [14, 15] with modules and parametric modules being eliminated at compile time through static interpretation. Modules can be used to encode certain classes of higher-order functions, but they are quite restricted compared to real higher-order functions and do not allow for the representation of first-class functions in general.

Futhark has a heavily optimizing compiler that generates OpenCL code optimized for GPU execution. The compiler uses a first-order intermediate language representation, which provides another motivation for removing higher-order functions early on, so that we do not have to modify a large part of the existing compilation pipeline and since many optimizations are likely simpler to perform on a first-order language. Furthermore, the OpenCL interface does not allow function pointers.

Chapter 3

Language

To be able to formally define and reason about the defunctionalization transformation, to be presented in Chapter 4, we define a simple functional language on which the transformation will operate. Conceptually, the transformation goes from a source language to a target language, but since the target language will be a sublanguage of the source language, we shall generally treat them as one and the following definitions will apply to both languages, unless stated otherwise.

The language is a λ-calculus extended with various features to resemble the Futhark language, including records, arrays with in-place updates, a parallel map, and a sequential loop construct. In the following, we define its abstract syntax, type system, and operational semantics.

3.1 Syntax

The set of *types* of the source language is given by the following grammar. The metavariable $\ell \in \mathbf{Lab}$ ranges over record *labels*.

$$\tau ::= \mathbf{int} \mid \mathbf{bool} \mid \tau_1 \to \tau_2 \mid \{(\ell_i : \tau_i)^{i \in 1..n}\} \mid []\tau$$

Record types are considered identical up to permutation of fields.

The abstract syntax of *expressions* of the source language is given by the following grammar. The metavariable $x \in \mathbf{Var}$ ranges over *variables* of the source language. We assume an injective function $Lab : \mathbf{Var} \to \mathbf{Lab}$ that maps variables to labels. Additionally, we let $n \in \mathbb{Z}$.

$$\begin{aligned} e ::= {} & x \mid \overline{n} \mid \mathbf{true} \mid \mathbf{false} \mid e_1 + e_2 \mid e_1 \leq e_2 \\ & \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \\ & \mid \lambda x : \tau.\, e_0 \mid e_1\ e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \\ & \mid \{(\ell_i = e_i)^{i \in 1..n}\} \mid e_0.\ell \\ & \mid [(e_i)^{i \in 1..n}] \mid e_0[e_1] \mid e_0\ \mathbf{with}\ [e_1] \leftarrow e_2 \mid \mathbf{length}\ e_0 \\ & \mid \mathbf{map}\ (\lambda x.\, e_1)\ e_2 \\ & \mid \mathbf{loop}\ x = e_0\ \mathbf{for}\ y\ \mathbf{in}\ e_1\ \mathbf{do}\ e_2 \end{aligned}$$

Expressions are considered identical up to renaming of bound variables. Array literals are required to be non-empty in order to simplify the rules and relations in the following and in the metatheory. Empty arrays can be supported fairly easily, for example by annotating arrays with the type of their elements.

The syntax of expressions of the target language is identical to that of the source language except that it does not have λ-abstractions and application. Similarly, the types of the target language does not include function types.[1]

We define a judgment, τ orderZero, given by the rules in Figure 3.1, asserting that a type τ has order zero, which means that τ does not contain any function type as a subterm.

$\boxed{\tau \,\text{orderZero}}$

$$\frac{}{\mathbf{int}\,\text{orderZero}} \qquad \frac{}{\mathbf{bool}\,\text{orderZero}}$$

$$\frac{(\tau_i\,\text{orderZero})^{i\in 1..n}}{\{(\ell_i : \tau_i)^{i\in 1..n}\}\,\text{orderZero}} \qquad \frac{\tau\,\text{orderZero}}{[\,]\tau\,\text{orderZero}}$$

Figure 3.1: Judgment asserting that a type has order zero.

3.2 Type system

The typing rules for the language are mostly standard except for restrictions on the use of functions in certain places. Specifically, a conditional may not return a function, arrays are not allowed to contain functions, and a loop may not produce a function. These restrictions are enforced by the added premise of the judgment τ orderZero in the rules for conditionals, array literals, parallel maps, and loops. Aside from these restrictions, the use of higher-order functions and functions as first-class values is not restricted and, in particular, records are allowed to contain functions of arbitrarily high order. It is worth emphasizing that we only restrict the form of the results produced by conditionals and loops, and the results of expressions contained in arrays; the subexpressions themselves may contain definitions and applications of arbitrary functions.

A *typing context* (or *type environment*) Γ is a finite sequence of variables associated with their types:

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

The empty context is denoted by $\cdot$, but is often omitted from the actual judgments. The variables in a typing context are required to be distinct. This requirement can always be satisfied by renaming bound variables as necessary.

The set of variables bound in a typing context is denoted by $\operatorname{dom}\Gamma$ and the type of a variable x bound in Γ is denoted by $\Gamma(x)$ if it exists. We write Γ, Γ' to denote the typing context consisting of the mappings in Γ followed by the mappings in Γ'. Note that since the variables in a context are distinct, the ordering is insignificant. Additionally, we write $\Gamma \subseteq \Gamma'$ if $\Gamma'(x) = \Gamma(x)$ for all $x \in \operatorname{dom}\Gamma$.

[1] In the actual implementation, the target language does include application of first-order functions, but in our theoretical work we just inline the functions for simplicity.

The typing rules for the language are given in Figure 3.2.

$\boxed{\Gamma \vdash e : \tau}$

$$\textsc{T-Var}: \frac{}{\Gamma \vdash x : \tau}\ (\Gamma(x) = \tau) \qquad \textsc{T-Num}: \frac{}{\Gamma \vdash \overline{n} : \mathbf{int}}$$

$$\textsc{T-True}: \frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}} \qquad \textsc{T-False}: \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}}$$

$$\textsc{T-Plus}: \frac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}} \qquad \textsc{T-Leq}: \frac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 \leq e_2 : \mathbf{bool}}$$

$$\textsc{T-If}: \frac{\Gamma \vdash e_1 : \mathbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau \qquad \tau\,\mathrm{orderZero}}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}$$

$$\textsc{T-Lam}: \frac{\Gamma, x : \tau_1 \vdash e_0 : \tau_2}{\Gamma \vdash \lambda x\colon \tau_1.\, e_0 : \tau_1 \to \tau_2} \qquad \textsc{T-App}: \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\textsc{T-Let}: \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\textsc{T-Rcd}: \frac{(\Gamma \vdash e_i : \tau_i)^{i \in 1..n}}{\Gamma \vdash \{(\ell_i = e_i)^{i \in 1..n}\} : \{(\ell_i : \tau_i)^{i \in 1..n}\}}$$

$$\textsc{T-Proj}: \frac{\Gamma \vdash e_0 : \{(\ell_i : \tau_i)^{i \in 1..n}\}}{\Gamma \vdash e_0.\ell_k : \tau_k}\ (1 \leq k \leq n)$$

$$\textsc{T-Array}: \frac{(\Gamma \vdash e_i : \tau)^{i \in 1..n} \qquad \tau\,\mathrm{orderZero}}{\Gamma \vdash [e_1, \ldots, e_n] : [\,]\tau}$$

$$\textsc{T-Index}: \frac{\Gamma \vdash e_0 : [\,]\tau \qquad \Gamma \vdash e_1 : \mathbf{int}}{\Gamma \vdash e_0[e_1] : \tau}$$

$$\textsc{T-Update}: \frac{\Gamma \vdash e_0 : [\,]\tau \qquad \Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_0\ \mathbf{with}\ [e_1] \leftarrow e_2 : [\,]\tau}$$

$$\textsc{T-Length}: \frac{\Gamma \vdash e_0 : [\,]\tau}{\Gamma \vdash \mathbf{length}\ e_0 : \mathbf{int}}$$

$$\textsc{T-Map}: \frac{\Gamma \vdash e_2 : [\,]\tau_2 \qquad \Gamma, x : \tau_2 \vdash e_1 : \tau \qquad \tau\,\mathrm{orderZero}}{\Gamma \vdash \mathbf{map}\ (\lambda x.\, e_1)\ e_2 : [\,]\tau}$$

$$\textsc{T-Loop}: \frac{\Gamma \vdash e_0 : \tau \qquad \Gamma \vdash e_1 : [\,]\tau' \qquad \Gamma, x : \tau, y : \tau' \vdash e_2 : \tau \qquad \tau\,\mathrm{orderZero}}{\Gamma \vdash \mathbf{loop}\ x = e_0\ \mathbf{for}\ y\ \mathbf{in}\ e_1\ \mathbf{do}\ e_2 : \tau}$$

Figure 3.2: Typing rules.

Note that we do not require the annotated type in λ-abstractions to be well-formed, that is, to not contain any function types within array types. This is not an issue for the translation or the metatheory, since such a function will still translate to a well-typed expression of order zero and if the function is ever applied, then the typing rules ensure that the expression is not well-typed.

3.3 Operational semantics

The operational semantics of the source (and target) language could be defined in a completely standard way, but for the sake of the metatheory and the connection with the defunctionalization transformation to be presented later, we choose to define an operational semantics with an evaluation environment and function closures rather than using simple β-reduction for evaluation of applications. The semantics is given in a big-step style for the same reasons.

Evaluation environments Σ and values v are defined mutually inductively as follows. A function *closure* is a value that captures the environment in which a λ-abstraction was evaluated and is denoted by:

$$\mathit{clos}(\lambda x\colon \tau.\, e_0, \Sigma)$$

The *values* of the source language are as follows:

$$v ::= \overline{n} \mid \mathbf{true} \mid \mathbf{false} \mid \mathit{clos}(\lambda x\colon \tau.\, e_0, \Sigma) \mid \{(\ell_i = v_i)^{i \in 1..n}\} \mid [(v_i)^{i \in 1..n}]$$

The values of the target language are the same, but without function closures. An *evaluation environment* Σ is a mapping from variables to values and has the same properties and notations as the typing context with regards to extension, variable lookup, and distinctness of variables:

$$\Sigma ::= \cdot \mid \Sigma, x \mapsto v$$

Because the language involves array indexing and updating, a well-typed program may still not evaluate to one of the above values, in case an attempt is made to access an index outside the bounds of an array. To be able to distinguish between such an out-of-bounds error and a stuck expression that is neither a value nor can evaluate to anything, we introduce the special term **err** to denote an out-of-bounds error and we define a *result* r to be either a value or **err**:

$$r ::= v \mid \mathbf{err}$$

The big-step operational semantics for the language is given by the derivation rules in Figure 3.3. In case any subexpression evaluates to **err**, the entire expression should evaluate to **err**, so it is necessary to give derivation rules for propagating these error results. Unfortunately, this error propagation involves creating many extra derivation rules and duplicating many premises. We show the rules that introduce **err**; however, we choose to omit the ones that propagate errors and instead just note that for each of the non-axiom rules below, there are a number of additional rules for propagating errors. For instance, for the rule E-App, there are additional rules E-AppErr$\{1, 2, 0\}$, which propagate errors in the applied expression, the argument, and the closure body, respectively.

The rule E-Loop refers to an auxiliary judgment form, defined in Figure 3.4, which performs the iterations of the loop, given a starting value and a sequence of values to iterate over. Like the main evaluation judgment, this one also has rules for propagating **err** results, which are again omitted.

$\boxed{\Sigma \vdash e \downarrow r}$

$$\textsc{E-Var}: \frac{}{\Sigma \vdash x \downarrow v}\ (\Sigma(x) = v) \qquad \textsc{E-Num}: \frac{}{\Sigma \vdash \overline{n} \downarrow \overline{n}} \qquad \textsc{E-True}: \frac{}{\Sigma \vdash \mathbf{true} \downarrow \mathbf{true}}$$

$$\textsc{E-Plus}: \frac{\Sigma \vdash e_1 \downarrow \overline{n_1} \qquad \Sigma \vdash e_2 \downarrow \overline{n_2}}{\Sigma \vdash e_1 + e_2 \downarrow \overline{n_1 + n_2}} \qquad \textsc{E-False}: \frac{}{\Sigma \vdash \mathbf{false} \downarrow \mathbf{false}}$$

$$\textsc{E-LeqT}: \frac{\Sigma \vdash e_1 \downarrow \overline{n_1} \quad \Sigma \vdash e_2 \downarrow \overline{n_2}}{\Sigma \vdash e_1 \leq e_2 \downarrow \mathbf{true}}\ (n_1 \leq n_2) \qquad \textsc{E-LeqF}: \frac{\Sigma \vdash e_1 \downarrow \overline{n_1} \quad \Sigma \vdash e_2 \downarrow \overline{n_2}}{\Sigma \vdash e_1 \leq e_2 \downarrow \mathbf{false}}\ (n_1 > n_2)$$

$$\textsc{E-IfT}: \frac{\Sigma \vdash e_1 \downarrow \mathbf{true} \qquad \Sigma \vdash e_2 \downarrow v}{\Sigma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \downarrow v} \qquad \textsc{E-IfF}: \frac{\Sigma \vdash e_1 \downarrow \mathbf{false} \qquad \Sigma \vdash e_3 \downarrow v}{\Sigma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \downarrow v}$$

$$\textsc{E-Lam}: \frac{}{\Sigma \vdash \lambda x\colon \tau.\, e_0 \downarrow \mathit{clos}(\lambda x\colon \tau.\, e_0, \Sigma)}$$

$$\textsc{E-App}: \frac{\Sigma \vdash e_1 \downarrow \mathit{clos}(\lambda x\colon \tau.\, e_0, \Sigma_0) \quad \Sigma \vdash e_2 \downarrow v_2 \quad \Sigma_0, x \mapsto v_2 \vdash e_0 \downarrow v}{\Sigma \vdash e_1\ e_2 \downarrow v} \qquad \textsc{E-Let}: \frac{\Sigma \vdash e_1 \downarrow v_1 \quad \Sigma, x \mapsto v_1 \vdash e_2 \downarrow v}{\Sigma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \downarrow v}$$

$$\textsc{E-Rcd}: \frac{(\Sigma \vdash e_i \downarrow v_i)^{i \in 1..n}}{\Sigma \vdash \{(\ell_i = e_i)^{i \in 1..n}\} \downarrow \{(\ell_i = v_i)^{i \in 1..n}\}}$$

$$\textsc{E-Proj}: \frac{\Sigma \vdash e_0 \downarrow \{(\ell_i = v_i)^{i \in 1..n}\}}{\Sigma \vdash e_0.\ell_k \downarrow v_k}\ (1 \leq k \leq n)$$

$$\textsc{E-Array}: \frac{(\Sigma \vdash e_i \downarrow v_i)^{i \in 1..n}}{\Sigma \vdash [(e_i)^{i \in 1..n}] \downarrow [(v_i)^{i \in 1..n}]} \qquad \textsc{E-Index}: \frac{\Sigma \vdash e_0 \downarrow [(v_i)^{i \in 1..n}] \quad \Sigma \vdash e_1 \downarrow \overline{k}}{\Sigma \vdash e_0[e_1] \downarrow v_k}\ (1 \leq k \leq n)$$

$$\textsc{E-IndexErr}: \frac{\Sigma \vdash e_0 \downarrow [(v_i)^{i \in 1..n}] \qquad \Sigma \vdash e_1 \downarrow \overline{k}}{\Sigma \vdash e_0[e_1] \downarrow \mathbf{err}}\ (k < 1 \vee k > n)$$

$$\textsc{E-Update}: \frac{\Sigma \vdash e_0 \downarrow [(v_i)^{i \in 1..n}] \qquad \Sigma \vdash e_1 \downarrow \overline{k} \qquad \Sigma \vdash e_2 \downarrow v'_k}{\Sigma \vdash e_0\ \mathbf{with}\ [e_1] \leftarrow e_2 \downarrow [(v_i)^{i \in 1..k-1}, v'_k, (v_i)^{i \in k+1..n}]}\ (1 \leq k \leq n)$$

$$\textsc{E-UpdateErr}: \frac{\Sigma \vdash e_0 \downarrow [(v_i)^{i \in 1..n}] \qquad \Sigma \vdash e_1 \downarrow \overline{k}}{\Sigma \vdash e_0\ \mathbf{with}\ [e_1] \leftarrow e_2 \downarrow \mathbf{err}}\ (k < 1 \vee k > n)$$

$$\textsc{E-Length}: \frac{\Sigma \vdash e_0 \downarrow [(v_i)^{i \in 1..n}]}{\Sigma \vdash \mathbf{length}\ e_0 \downarrow \overline{n}} \qquad \textsc{E-Map}: \frac{\Sigma \vdash e_2 \downarrow [(v'_i)^{i \in 1..n}] \quad (\Sigma, x \mapsto v'_i \vdash e_1 \downarrow v_i)^{i \in 1..n}}{\Sigma \vdash \mathbf{map}\ (\lambda x.\, e_1)\ e_2 \downarrow [(v_i)^{i \in 1..n}]}$$

$$\textsc{E-Loop}: \frac{\Sigma \vdash e_0 \downarrow v_0 \qquad \Sigma \vdash e_1 \downarrow [(v_i)^{i \in 1..n}] \quad \Sigma; x = v_0; y = (v_i)^{i \in 1..n} \vdash e_2 \downarrow v}{\Sigma \vdash \mathbf{loop}\ x = e_0\ \mathbf{for}\ y\ \mathbf{in}\ e_1\ \mathbf{do}\ e_2 \downarrow v}$$

Figure 3.3: Big-step operational semantics.

$$\boxed{\Sigma; x = v_0; y = (v_i)^{i \in 1..n} \vdash e \downarrow r}$$

$$\text{EL-Nil:}\ \frac{}{\Sigma; x = v_0; y = \cdot \vdash e \downarrow v_0}$$

$$\text{EL-Cons:}\ \frac{\Sigma, x \mapsto v_0, y \mapsto v_1 \vdash e \downarrow v_0' \qquad \Sigma; x = v_0'; y = (v_i)^{i \in 2..n} \vdash e \downarrow v}{\Sigma; x = v_0; y = (v_i)^{i \in 1..n} \vdash e \downarrow v}$$

Figure 3.4: Auxiliary judgment for the semantics of loops.

Chapter 4

Defunctionalization

We now define the defunctionalization transformation which translates an expression in the source language to an equivalent expression in the target language that does not contain any higher-order subterms or use of first-class functions.

The definitions of translation environments and static values are given by mutually inductive definitions as follows. A *translation environment* (or *defunctionalization environment*) E is a finite sequence of mappings from variables to static values:

$$E ::= \cdot \mid E, x \mapsto sv$$

We assume the same properties as we did for typing contexts and evaluation environments, and we use analogous notation. *Static values* are defined as follows:

$$\begin{aligned} sv ::= \; & \mathit{Dyn}\ \tau \\ \mid \; & \mathit{Lam}\ x\ e_0\ E \\ \mid \; & \mathit{Rcd}\ \{(\ell_i \mapsto sv_i)^{i \in 1..n}\} \\ \mid \; & \mathit{Arr}\ sv_0 \end{aligned}$$

As the name suggests, a static value is essentially a static approximation of the value that an expression will eventually evaluate to. This resembles the role of types, which also approximate the values of expressions, but static values posses more information than types. As a result of the restrictions on the use of functions in the type system, the static value *Lam* that approximates functional values, will contain the actual function parameter and body, along with a defunctionalization environment containing static values approximating the values in the closed-over environment.

The defunctionalization translation takes place in a defunctionalization environment, as defined above, which mirrors the evaluation environment by approximating the values by static values, and it translates a given expression e to a *residual expression* e' and its corresponding static value sv. The residual expression resembles the original expression, but λ-abstractions are translated into record expressions that capture the values in the environment at the time of evaluation. Applications are translated into **let**-bindings that bind the record expression, the closed-over variables, and the function parameter.

As with record types, we consider *Rcd* static values to be identical up to reordering of the label-entries. Additionally, we consider *Lam* static values to be identical up to renaming of the parameter variable, as for λ-abstractions.

The transformation is defined by the derivation rules in Figure 4.1 and Figure 4.2.

$\boxed{E \vdash e \rightsquigarrow \langle e', sv \rangle}$

$$\textsc{D-Var}: \frac{}{E \vdash x \rightsquigarrow \langle x, sv \rangle}\ (E(x) = sv) \qquad \textsc{D-Num}: \frac{}{E \vdash \overline{n} \rightsquigarrow \langle \overline{n}, \mathit{Dyn}\ \mathbf{int} \rangle}$$

$$\textsc{D-True}: \frac{}{E \vdash \mathbf{true} \rightsquigarrow \langle \mathbf{true}, \mathit{Dyn}\ \mathbf{bool} \rangle} \quad \text{(equivalent rule D-False)}$$

$$\textsc{D-Plus}: \frac{\begin{array}{c} E \vdash e_1 \rightsquigarrow \langle e_1', \mathit{Dyn}\ \mathbf{int} \rangle \\ E \vdash e_2 \rightsquigarrow \langle e_2', \mathit{Dyn}\ \mathbf{int} \rangle \end{array}}{E \vdash e_1 + e_2 \rightsquigarrow \langle e_1' + e_2', \mathit{Dyn}\ \mathbf{int} \rangle} \quad \text{(similar rule D-Leq)}$$

$$\textsc{D-If}: \frac{\begin{array}{c} E \vdash e_1 \rightsquigarrow \langle e_1', \mathit{Dyn}\ \mathbf{bool} \rangle \\ E \vdash e_2 \rightsquigarrow \langle e_2', sv \rangle \qquad E \vdash e_3 \rightsquigarrow \langle e_3', sv \rangle \end{array}}{E \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rightsquigarrow \langle \mathbf{if}\ e_1'\ \mathbf{then}\ e_2'\ \mathbf{else}\ e_3', sv \rangle}$$

$$\textsc{D-Lam}: \frac{}{E \vdash \lambda x\colon \tau.\, e_0 \rightsquigarrow \left\langle \{(\mathit{Lab}(y) = y)^{y \in \mathrm{dom}\, E}\}, \mathit{Lam}\ x\ e_0\ E \right\rangle}$$

$$\textsc{D-App}: \frac{\begin{array}{c} E \vdash e_1 \rightsquigarrow \langle e_1', \mathit{Lam}\ x\ e_0\ E_0 \rangle \\ E \vdash e_2 \rightsquigarrow \langle e_2', sv_2 \rangle \qquad E_0, x \mapsto sv_2 \vdash e_0 \rightsquigarrow \langle e_0', sv \rangle \end{array}}{E \vdash e_1\ e_2 \rightsquigarrow \langle e', sv \rangle}$$

$$\text{where } e' = \begin{array}{l} \mathbf{let}\ \mathit{env} = e_1'\ \mathbf{in}\ (\mathbf{let}\ y = \mathit{env}.\mathit{Lab}(y)\ \mathbf{in})^{y \in \mathrm{dom}\, E_0} \\ \mathbf{let}\ x = e_2'\ \mathbf{in}\ e_0' \end{array}$$

$$\textsc{D-Let}: \frac{E \vdash e_1 \rightsquigarrow \langle e_1', sv_1 \rangle \qquad E, x \mapsto sv_1 \vdash e_2 \rightsquigarrow \langle e_2', sv \rangle}{E \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \langle \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2', sv \rangle}$$

Figure 4.1: Derivation rules for the defunctionalization transformation.

In the implementation, the record in the residual expression of rule D-Lam only captures the free variables in the λ-abstraction. Likewise, the defunctionalization environment embedded in the static value is restricted to the free variables. This refinement is not hard to formalize, but it does not add anything interesting to the development, so we have omitted it for simplicity.

Notice how the rules include aspects of both evaluation and type checking, in analogy to how static values are somewhere in-between values and types. For instance, the rules ensure that variables are in scope, and that a conditional has a *Dyn* Boolean condition and the branches have the same static value. Somewhat curiously, this constraint on the static values of branches actually allows for a conditional to return functions in its branches, as long as the functions are α-equivalent. The same is true for arrays and loops.

This transformation is able to translate any order zero expression into an equivalent expression that does not contain any higher-order functions. Any

$\boxed{E \vdash e \rightsquigarrow \langle e', sv\rangle}$

$$\text{D-Rcd:}\ \frac{(E \vdash e_i \rightsquigarrow \langle e_i', sv_i\rangle)^{i\in 1..n}}{E \vdash \{(\ell_i = e_i)^{i\in 1..n}\} \rightsquigarrow \left\langle \{(\ell_i = e_i')^{i\in 1..n}\}, Rcd\ \{(\ell_i \mapsto sv_i)^{i\in 1..n}\}\right\rangle}$$

$$\text{D-Proj:}\ \frac{E \vdash e_0 \rightsquigarrow \left\langle e_0', Rcd\ \{(\ell_i \mapsto sv_i)^{i\in 1..n}\}\right\rangle}{E \vdash e_0.\ell_k \rightsquigarrow \langle e_0'.\ell_k, sv_k\rangle}\ (1 \le k \le n)$$

$$\text{D-Array:}\ \frac{(E \vdash e_i \rightsquigarrow \langle e_i', sv\rangle)^{i\in 1..n}}{E \vdash [e_1, \ldots, e_n] \rightsquigarrow \langle [e_1', \ldots, e_n'], Arr\ sv\rangle}$$

$$\text{D-Index:}\ \frac{E \vdash e_1 \rightsquigarrow \langle e_1', Arr\ sv\rangle \qquad E \vdash e_2 \rightsquigarrow \langle e_2', Dyn\ \mathbf{int}\rangle}{E \vdash e_1[e_2] \rightsquigarrow \langle e_1'[e_2'], sv\rangle}$$

$$\text{D-Update:}\ \frac{\begin{array}{c}E \vdash e_0 \rightsquigarrow \langle e_0', Arr\ sv\rangle \\ E \vdash e_1 \rightsquigarrow \langle e_1', Dyn\ \mathbf{int}\rangle \qquad E \vdash e_2 \rightsquigarrow \langle e_2', sv\rangle\end{array}}{E \vdash e_0\ \mathbf{with}\ [e_1] \leftarrow e_2 \rightsquigarrow \langle e_0'\ \mathbf{with}\ [e_1'] \leftarrow e_2', Arr\ sv\rangle}$$

$$\text{D-Length:}\ \frac{E \vdash e_0 \rightsquigarrow \langle e_0', Arr\ sv\rangle}{E \vdash \mathbf{length}\ e_0 \rightsquigarrow \langle \mathbf{length}\ e_0', Dyn\ \mathbf{int}\rangle}$$

$$\text{D-Map:}\ \frac{E \vdash e_2 \rightsquigarrow \langle e_2', Arr\ sv_2\rangle \qquad E, x \mapsto sv_2 \vdash e_1 \rightsquigarrow \langle e_1', sv_1\rangle}{E \vdash \mathbf{map}\ (\lambda x.\ e_1)\ e_2 \rightsquigarrow \langle \mathbf{map}\ (\lambda x.\ e_1')\ e_2', Arr\ sv_1\rangle}$$

$$\text{D-Loop:}\ \frac{\begin{array}{c}E \vdash e_0 \rightsquigarrow \langle e_0', sv\rangle \qquad E \vdash e_1 \rightsquigarrow \langle e_1', Arr\ sv_1\rangle \\ E, x \mapsto sv, y \mapsto sv_1 \vdash e_2 \rightsquigarrow \langle e_2', sv\rangle\end{array}}{\begin{array}{l}E \vdash \mathbf{loop}\ x = e_0\ \mathbf{for}\ y\ \mathbf{in}\ e_1\ \mathbf{do}\ e_2 \\ \quad \rightsquigarrow \langle \mathbf{loop}\ x = e_0'\ \mathbf{for}\ y\ \mathbf{in}\ e_1'\ \mathbf{do}\ e_2', sv\rangle\end{array}}$$

Figure 4.2: Derivation rules for the defunctionalization transformation (cont.).

first-order expression can be translated by converting the types of its parameters (which are necessarily order zero) to static values and including these as bindings for the parameter variables in an initial translation environment. This conversion is straightforwardly defined as follows:

$$\begin{aligned}\overline{\mathbf{int}} &= Dyn\ \mathbf{int} \\ \overline{\mathbf{bool}} &= Dyn\ \mathbf{bool} \\ \overline{\{(\ell_i\colon \tau_i)^{i\in 1..n}\}} &= Rcd\ \{(\ell_i \mapsto \overline{\tau_i})^{i\in 1..n}\} \\ \overline{[\,]\tau} &= Arr\ \overline{\tau}\end{aligned}$$

By a relatively simple extension to the system that has been presented so far, it is possible to support any number of top-level function definitions that take parameters of arbitrary type and can have any return type, as long as the designated *main* function is first-order.

Chapter 5

Metatheory

In this chapter, we show type soundness and argue for the correctness of the defunctionalization transformation presented in Chapter 4. We show that the transformation of a well-typed expression always terminates and yields another well-typed expression. Finally, we show that the meaning of a defunctionalized expression is consistent with the meaning of the original expression.

5.1 Type soundness and normalization

We first show type soundness. Since we are using a big-step semantics, the situation is a bit different from the usual approach of showing progress and preservation for a small-step semantics. One of the usual advantages of using a small-step semantics is that it allows distinguishing between diverging and stuck terms, whereas for a big-step semantics, neither a diverging term nor a stuck term is related to any value. As we shall see, however, for the big-step semantics that we have presented, any well-typed expression will evaluate to a result that is either **err** or a value that is, semantically, of the same type. Thus, we simultaneously establish that the language is strongly normalizing, which comes as no surprise given the lack of recursion and the bounded number of iterations of loops.

To this end, we first define a relation between values and types, given by derivation rules in Figure 5.1, and extend it to relate evaluation environments and typing contexts. We can also view this relation as a logical predicate on values indexed by types, or as a semantic typing judgment on values.

We then state and prove type soundness as follows. We only show a couple of cases. The proof of termination and preservation of typing for defunctionalization is fairly similar and we go into more detail in that proof, in Section 5.2.

Lemma 1 (Type soundness)**.** *If* $\Gamma \vdash e : \tau$ *(by* $\mathcal{T}$*) and* $\vDash \Sigma : \Gamma$ *(by* $\mathcal{R}$*), for some* Σ*, then* $\Sigma \vdash e \downarrow r$*, for some* r*, and either* $r = \mathbf{err}$ *or* $r = v$*, for some* v*, and* $\vDash v : \tau$*.*

Proof. By induction on the typing derivation $\mathcal{T}$.

In most cases we simply apply the induction hypothesis to each subderivation, in turn, and reason on whether the result is **err** or a value. In multiple cases, we relate an extended typing context and evaluation environment using

$\boxed{\vDash v : \tau}$

$$\frac{}{\vDash \overline{n} : \mathbf{int}} \qquad \frac{}{\vDash \mathbf{true} : \mathbf{bool}} \qquad \frac{}{\vDash \mathbf{false} : \mathbf{bool}}$$

$$\frac{\forall v_1.\ \vDash v_1 : \tau_1 \implies \exists r.\ \Sigma, x \mapsto v_1 \vdash e_0 \downarrow r \wedge (r = \mathbf{err} \vee (r = v_2 \wedge \vDash v_2 : \tau_2))}{\vDash clos(\lambda x\colon \tau_1.\, e_0, \Sigma) : \tau_1 \to \tau_2}$$

$$\frac{(\vDash v_i : \tau_i)^{i \in 1..n}}{\vDash \{(\ell_i = v_i)^{i \in 1..n}\} : \{(\ell_i : \tau_i)^{i \in 1..n}\}} \qquad \frac{(\vDash v_i : \tau)^{i \in 1..n}}{\vDash [(v_i)^{i \in 1..n}] : [\,]\tau}$$

$\boxed{\vDash \Sigma : \Gamma}$

$$\frac{}{\vDash \cdot : \cdot} \qquad \frac{\vDash \Sigma : \Gamma \qquad \vDash v : \tau}{\vDash (\Sigma, x \mapsto v) : (\Gamma, x : \tau)}$$

Figure 5.1: Relation between values and types, and evaluation environments and typing contexts, respectively.

assumption $\mathcal{R}$ and relations obtained from the induction hypothesis, to allow for further applications of the induction hypothesis.

In the case for T-Loop, in the subcase where the first two subexpressions evaluate to a value and an array of values, respectively, we proceed by an inner induction on the structure of the corresponding sequence of values for the loop. If the sequence is empty, we get the necessary derivation of the auxiliary judgment for loop iteration directly by axiom. In the inductive case, we apply the outer induction hypothesis on the subderivation for the loop body. If the result is **err**, it is again just propagated. If it is a value, we can relate the value with the type of the loop and then apply the inner induction hypothesis.

We show just a few representative cases.

- Case $\mathcal{T} = \dfrac{}{\Gamma \vdash x : \tau}\,(\Gamma(x) = \tau)$.

  Since $\vDash \Sigma : \Gamma$ and $\Gamma(x) = \tau$, we must have a subderivation of $\vDash v : \tau$, for some $\Sigma(x) = v$. Then by rule E-Var, $\Sigma \vdash x \downarrow v$, as required.

- Case $\mathcal{T} = \dfrac{\begin{array}{c}\mathcal{T}_0 \\ \Gamma, x : \tau_1 \vdash e_0 : \tau_2\end{array}}{\Gamma \vdash \lambda x\colon \tau_1.\, e_0 : \tau_1 \to \tau_2}$.

  By rule E-Lam, we have that $\Sigma \vdash \lambda x\colon \tau_1.\, e_0 \downarrow clos(\lambda x\colon \tau_1.\, e_0, \Sigma)$.

  Now, assume some value v_1 such that $\vDash v_1 : \tau_1$. Since $\vDash \Sigma : \Gamma$ by assumption $\mathcal{R}$, we have by definition also $\vDash (\Sigma, x \mapsto v_1) : (\Gamma, x : \tau_1)$. Then by the induction hypothesis on $\mathcal{T}_0$ with this, we get $\Sigma, x \mapsto v_1 \vdash e_0 \downarrow r_0$ and either $r_0 = \mathbf{err}$, or $r_0 = v_0$ and $\vDash v_0 : \tau_2$. Thus, by definition of the relation, we get $\vDash clos(\lambda x\colon \tau_1.\, e_0, \Sigma) : \tau_1 \to \tau_2$.

- Case $\mathcal{T} = \dfrac{\begin{array}{cc}\mathcal{T}_1 & \mathcal{T}_2 \\ \Gamma \vdash e_1 : \tau_2 \to \tau & \Gamma \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash e_1\ e_2 : \tau}$.

  By the induction hypothesis on $\mathcal{T}_1$ with $\mathcal{R}$, we get $\Sigma \vdash e_1 \downarrow r_1$ and either $r_1 = \mathbf{err}$ or $r_1 = v_1$, for some v_1 with $\vDash v_1 : \tau_2 \to \tau$. In the former case, $\Sigma \vdash e_1\ e_2 \downarrow \mathbf{err}$ by E-AppErr1. In the latter case, by

inversion we must have that $v_1 = \mathit{clos}(\lambda x\colon \tau_2.\ e_0, \Sigma_0)$ for some x, e_0, and Σ_0, and by definition of the relation, if $\vDash v' : \tau_2$ for any v', then $\exists r.\ \Sigma_0, x \mapsto v' \vdash e_0 \downarrow r$ and either $r = \mathbf{err}$ or $r = v$ with $\vDash v : \tau$.

By the induction hypothesis on $\mathcal{T}_2$ with $\mathcal{R}$, we get $\Sigma \vdash e_2 \downarrow r_2$ and either $r_2 = \mathbf{err}$ or $r_2 = v_2$ with $\vDash v_2 : \tau_2$. In the former case, $\Sigma \vdash e_1\ e_2 \downarrow \mathbf{err}$ by E-AppErr2. In the latter case, by the implication from previously, we get that there exists r such that $\Sigma_0, x \mapsto v_2 \vdash e_0 \downarrow r$ and either $r = \mathbf{err}$ or $r = v$ for some v with $\vDash v : \tau$. In the former case, $\Sigma \vdash e_1\ e_2 \downarrow \mathbf{err}$ by E-AppErr0. In the latter, we get $\Sigma \vdash e_1\ e_2 \downarrow v$ by rule E-App.

- Case $\mathcal{T} = \dfrac{\overset{\mathcal{T}_2}{\Gamma \vdash e_2 : [\,]\tau_2} \qquad \overset{\mathcal{T}_1}{\Gamma, x : \tau_2 \vdash e_1 : \tau} \qquad \tau\ \mathrm{orderZero}}{\Gamma \vdash \mathbf{map}\ (\lambda x.\ e_1)\ e_2 : [\,]\tau}$.

  By the induction hypothesis on $\mathcal{T}_2$ with $\mathcal{R}$, we get $\Sigma \vdash e_2 \downarrow r_2$ and either $r_2 = \mathbf{err}$, or $r_2 = v_2$ with $\vDash v_2 : [\,]\tau_2$. In the former case, also $\Sigma \vdash \mathbf{map}\ (\lambda x.\ e_1)\ e_2 \downarrow \mathbf{err}$. In the latter case, by inversion on the relation, $v_2 = [v'_1, ..., v'_n]$ and $\vDash v'_i : \tau_2$, for each $i \in 1..n$.

  For each $i \in 1..n$, we construct $\vDash (\Sigma, x \mapsto v'_i) : (\Gamma, x : \tau_2)$ and by the induction hypothesis on $\mathcal{T}_1$ with this, we get $\Sigma, x \mapsto v'_i \vdash e_1 \downarrow r_i$. If $r_i = \mathbf{err}$ for any i, then $\Sigma \vdash \mathbf{map}\ (\lambda x.\ e_1)\ e_2 \downarrow \mathbf{err}$. Otherwise, if all $r_i = v''_i$, for some v''_i with $\vDash v''_i : \tau$, then $\Sigma \vdash \mathbf{map}\ (\lambda x. e_1)\ e_2 \downarrow [(v''_i)^{i \in 1..n}]$ and we construct $\vDash [(v''_i)^{i \in 1..n}] : [\,]\tau$.

□

5.2 Translation termination and preservation of typing

In this section, we show that the translation of a well-typed expression always terminates and that the translated expression is also well-typed, in a typing context that can be obtained from the defunctionalization environment and with a type that can be obtained from the static value.

We first define a mapping from static values to types, which shows how the type of a residual expression can be obtained from its static value, as will become evident later:

$$\begin{aligned}
[\![\, \mathit{Dyn}\ \tau \,]\!]_{\mathrm{tp}} &= \tau \\
[\![\, \mathit{Lam}\ x\ e_0\ E \,]\!]_{\mathrm{tp}} &= \{(\mathit{Lab}(y) : [\![\, sv_y \,]\!]_{\mathrm{tp}})^{(y \mapsto sv_y) \in E}\} \\
[\![\, \mathit{Rcd}\ \{(\ell_i \mapsto sv_i)^{i \in 1..n}\} \,]\!]_{\mathrm{tp}} &= \{(\ell_i : [\![\, sv_i \,]\!]_{\mathrm{tp}})^{i \in 1..n}\} \\
[\![\, \mathit{Arr}\ sv \,]\!]_{\mathrm{tp}} &= [\,]([\![\, sv \,]\!]_{\mathrm{tp}})
\end{aligned}$$

This is extended to map defunctionalization environments to typing contexts, by mapping each individual static value in an environment:

$$\begin{aligned}
[\![\, \cdot \,]\!]_{\mathrm{tp}} &= \cdot \\
[\![\, E, x \mapsto sv \,]\!]_{\mathrm{tp}} &= [\![\, E \,]\!]_{\mathrm{tp}}, x : [\![\, sv \,]\!]_{\mathrm{tp}}
\end{aligned}$$

In order to be able to show termination and preservation of typing for defunctionalization, we first define a relation, $\vDash sv : \tau$, between static values and

types, similar to the previous relation between values and types, and further extend it to relate defunctionalization environments and typing contexts. This relation is given by the rules in Figure 5.2.

$\boxed{\vDash sv : \tau}$

$$\frac{}{\vDash Dyn\ \mathbf{int} : \mathbf{int}} \qquad \frac{}{\vDash Dyn\ \mathbf{bool} : \mathbf{bool}}$$

$$\frac{\begin{array}{r}\forall sv_1.\ \vDash sv_1 : \tau_1 \implies \exists e_0', sv_2.\ E_0, x \mapsto sv_1 \vdash e_0 \rightsquigarrow \langle e_0', sv_2 \rangle \\ \wedge\ \vDash sv_2 : \tau_2\ \wedge\ [\![E_0, x \mapsto sv_1]\!]_{\mathrm{tp}} \vdash e_0' : [\![sv_2]\!]_{\mathrm{tp}}\end{array}}{\vDash Lam\ x\ e_0\ E_0 : \tau_1 \to \tau_2}$$

$$\frac{(\vDash sv_i : \tau_i)^{i \in 1..n}}{\vDash Rcd\ \{(\ell_i \mapsto sv_i)^{i \in 1..n}\} : \{(\ell_i : \tau_i)^{i \in 1..n}\}} \qquad \frac{\vDash sv : \tau \qquad \tau\ \mathrm{orderZero}}{\vDash Arr\ sv : [\,]\tau}$$

$\boxed{\vDash E : \Gamma}$

$$\frac{}{\vDash \cdot : \cdot} \qquad \frac{\vDash E : \Gamma \qquad \vDash sv : \tau}{\vDash (E, x \mapsto sv) : (\Gamma, x : \tau)}$$

Figure 5.2: Relation between static values and types, and defunctionalization environments and typing contexts, respectively.

By assuming this relation between some defunctionalization environment E and a typing context Γ for a given typing derivation, we can show that a well-typed expression will translate to some expression and additionally produce a static value that is related to the type of the original expression according to the above relation. Additionally, the translated expression is well-typed in the typing context obtained from E with a type determined by the static value. This strengthens the induction hypothesis to allow the case for application to go through, which would otherwise not be possible. This approach is quite similar to the previous proof of type soundness and normalization of evaluation.

In order to be able to prove the main result in Theorem 1, we state a number of lemmas in the following.

We first prove an auxiliary lemma about the above relation between static values and types, which states that for types of order zero, the related static value is uniquely determined. This property is crucial to the ability of defunctionalization to uniquely determine the function at every application site, and it is used in the proof of Theorem 1 in the cases for conditionals, array literals, array updates, and loops.

Lemma 2. *If* $\vDash sv : \tau$, $\vDash sv' : \tau$, *and* $\tau\,\mathrm{orderZero}$, *then* $sv = sv'$.

Proof. By induction on the derivation of $\vDash sv : \tau$. □

The following lemma states that if a static value is related to a type of order zero, then the static value maps to the same type. This property is used in Corollary 1 to establish that the types of order zero terms are unchanged by defunctionalization. It is also used in the cases for conditionals, array literals, loops, and maps in the proof of Theorem 1.

Lemma 3. *If* $\vDash sv : \tau$ *and* $\tau\,\mathrm{orderZero}$, *then* $[\![sv]\!]_{\mathrm{tp}} = \tau$.

Proof. By induction on the structure of sv. □

We then state the usual weakening lemma for typing derivations.

Lemma 4 (Weakening)**.** *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \subseteq \Gamma'$*, then also* $\Gamma' \vdash e : \tau$*.*

Proof. By induction on the structure of the expression e. □

Using this, we show the following lemma which allows a sequence of assumptions in a typing context to be "folded" into an assumption of a record type variable, containing the same types, where the expression in turn "unfolds" the variables by a sequence of nested **let**-bindings.

Lemma 5. *If* $\Gamma, \Gamma_0 \vdash e : \tau$*, then*

$$\Gamma, env : \{(Lab(x) : \tau_x)^{(x:\tau_x)\in\Gamma_0}\} \vdash (\textbf{let } x = env.Lab(x) \textbf{ in})^{x\in\operatorname{dom}\Gamma_0} \ \ e : \tau \ ,$$

where env *is a fresh variable not in* $\operatorname{dom}\Gamma, \Gamma_0$*.*

Proof. By Lemma 4 on the assumed typing derivation, we get a derivation of $\Gamma, env : \{(Lab(x) : \tau_x)^{(x:\tau_x)\in\Gamma_0}\}, \Gamma_0 \vdash e : \tau$. We then proceed by induction on the shape of Γ_0. □

The following lemma is used in the case for application in the proof of type preservation and termination, and is extracted into its own lemma in order to simplify the main proof.

Lemma 6. *If*

$$\begin{aligned}
&\mathcal{T}_1 :: \Gamma \vdash e_1 : [\![\, Lam\ x\ e_0\ E_0 \,]\!]_{\mathrm{tp}} \,, \\
&\mathcal{T}_2 :: \Gamma \vdash e_2 : \tau_2, \ \textit{and} \\
&\mathcal{T}_0 :: [\![\, E_0 \,]\!]_{\mathrm{tp}}, x : \tau_2 \vdash e_0 : \tau
\end{aligned}$$

then $\Gamma \vdash$ **let** $env = e_1$ **in** $(\textbf{let } y = env.Lab(y) \textbf{ in})^{y\in\operatorname{dom} E_0}$
let $x = e_2$ **in** $e_0 : \tau$.

Proof. By a direct proof.

By definition, $[\![\, Lam\ x\ e_0\ E_0 \,]\!]_{\mathrm{tp}} = \{(Lab(y) = [\![\, sv_y \,]\!]_{\mathrm{tp}})^{(y\mapsto sv_y)\in E_0}\}$. By weakening (Lemma 4) on $\mathcal{T}_0$ and by renaming of bound variables as necessary, we get $\mathcal{T}_0'$ of $\Gamma, [\![\, E_0 \,]\!]_{\mathrm{tp}}, x : \tau_2 \vdash e_0 : \tau$. By weakening on $\mathcal{T}_2$ and renaming as necessary, we get $\mathcal{T}_2'$ of $\Gamma, [\![\, E_0 \,]\!]_{\mathrm{tp}} \vdash e_2 : \tau_2$. By rule T-Let on $\mathcal{T}_2'$ and $\mathcal{T}_0'$, we get $\Gamma, [\![\, E_0 \,]\!]_{\mathrm{tp}} \vdash$ **let** $x = e_2$ **in** $e_0 : \tau$. By Lemma 5 on this, we get $\mathcal{T}'$ of:

$$\Gamma, env : \{(Lab(y) : \tau_y)^{(y\mapsto\tau_y)\in[\![\, E_0 \,]\!]_{\mathrm{tp}}}\} \vdash (\textbf{let } y = env.Lab(y) \textbf{ in})^{y\in\operatorname{dom}[\![\, E_0 \,]\!]_{\mathrm{tp}}} \textbf{let } x = e_2 \textbf{ in } e_0 : \tau$$

By definition, $\{(Lab(y) : \tau_y)^{(y\mapsto\tau_y)\in[\![\, E_0 \,]\!]_{\mathrm{tp}}}\} = \{(Lab(y) : [\![\, sv_y \,]\!]_{\mathrm{tp}})^{(y\mapsto sv_y)\in E_0}\}$ and $\operatorname{dom}[\![\, E_0 \,]\!]_{\mathrm{tp}} = \operatorname{dom} E_0$. Then by rule T-Let on $\mathcal{T}_1$ and $\mathcal{T}'$, we get the required derivation. □

Finally, we can state and prove termination and preservation of typing for the defunctionalization translation as follows:

Theorem 1. *If* $\Gamma \vdash e : \tau$ *(by* $\mathcal{T}$*) and* $\vDash E : \Gamma$ *(by* $\mathcal{R}$*), for some* E*, then* $E \vdash e \rightsquigarrow \langle e', sv \rangle$*,* $\vDash sv : \tau$*, and* $[\![E]\!]_{\text{tp}} \vdash e' : [\![sv]\!]_{\text{tp}}$*, for some* e' *and* sv*.*

Proof. By induction on the typing derivation $\mathcal{T}$.

We show a number of representative cases.

- Case $\mathcal{T} = \dfrac{}{\Gamma \vdash x : \tau}\,(\Gamma(x) = \tau)$.

  Since $\vDash E : \Gamma$, by the assumption $\mathcal{R}$, and $\Gamma(x) = \tau$, by the side condition, we must have that $E(x) = sv$, for some sv, and $\mathcal{R}$ must contain a subderivation of $\vDash sv : \tau$. By rule D-Var, we get $E \vdash x \rightsquigarrow \langle x, sv \rangle$. By definition, $[\![E]\!]_{\text{tp}}(x) = [\![sv]\!]_{\text{tp}}$ and then by rule T-Var, we get the required $[\![E]\!]_{\text{tp}} \vdash x : [\![sv]\!]_{\text{tp}}$.

- Case $\mathcal{T} = \dfrac{}{\Gamma \vdash \overline{n} : \mathbf{int}}$.

  By rule D-Num, $E \vdash \overline{n} \rightsquigarrow \langle \overline{n}, \mathit{Dyn}\ \mathbf{int} \rangle$. By axiom, $\vDash \mathit{Dyn}\ \mathbf{int} : \mathbf{int}$. By rule T-Num, $[\![E]\!]_{\text{tp}} \vdash \overline{n} : \mathbf{int}$ and by definition, $[\![\mathit{Dyn}\ \mathbf{int}]\!]_{\text{tp}} = \mathbf{int}$.

- The cases for T-True and T-False are analogous.

- Case $\mathcal{T} = \dfrac{\mathcal{Z} :: \tau\,\text{orderZero} \qquad \mathcal{T}_2 :: \Gamma \vdash e_2 : \tau \quad \mathcal{T}_1 :: \Gamma \vdash e_1 : \mathbf{bool} \qquad \mathcal{T}_3 :: \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}$.

  By the induction hypothesis on $\mathcal{T}_1$ with $\mathcal{R}$, we get derivations $\mathcal{D}_1$ of $E \vdash e_1 \rightsquigarrow \langle e_1', sv_1 \rangle$, $\mathcal{R}_1$ of $\vDash sv_1 : \mathbf{bool}$, and $\mathcal{T}_1'$ of $[\![E]\!]_{\text{tp}} \vdash e_1' : [\![sv_1]\!]_{\text{tp}}$. By inversion on $\mathcal{R}_1$, $sv_1 = \mathit{Dyn}\ \mathbf{bool}$, so by definition $[\![sv_1]\!]_{\text{tp}} = \mathbf{bool}$.

  For each $i \in \{2, 3\}$, by the induction hypothesis on $\mathcal{T}_i$ with $\mathcal{R}$, we get $\mathcal{D}_i$ of $E \vdash e_i \rightsquigarrow \langle e_i', sv_i \rangle$, $\mathcal{R}_i$ of $\vDash sv_i : \tau$, and $\mathcal{T}_i'$ of $[\![E]\!]_{\text{tp}} \vdash e_i' : [\![sv_i]\!]_{\text{tp}}$.

  By Lemma 2 on $\mathcal{R}_2$, $\mathcal{R}_3$, and $\mathcal{Z}$, we get that $sv_2 = sv_3$. Thus, by rule D-If on $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$, we get $E \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rightsquigarrow \langle \mathbf{if}\ e_1'\ \mathbf{then}\ e_2'\ \mathbf{else}\ e_3', sv_2 \rangle$ and we already know that $\vDash sv_2 : \tau$.

  By Lemma 3 on $\mathcal{R}_2$ and $\mathcal{Z}$, we have that $[\![sv_2]\!]_{\text{tp}} = \tau$. Then, by rule T-If on $\mathcal{T}_1'$, $\mathcal{T}_2'$, $\mathcal{T}_3'$, and $\mathcal{Z}$, we get the required typing derivation.

- The cases for T-Plus and T-Leq are similar.

- Case $\mathcal{T} = \dfrac{\mathcal{T}_0}{\Gamma \vdash \lambda x\colon \tau_1.\, e_0 : \tau_1 \to \tau_2}$.

  By rule D-Lam,

  $$E \vdash \lambda x\colon \tau_1.\, e_0 \rightsquigarrow \left\langle \{(\mathit{Lab}(y) = y)^{y \in \operatorname{dom} E}\}, \mathit{Lam}\ x\ e_0\ E \right\rangle,$$

  so $e' = \{(\mathit{Lab}(y) = y)^{y \in \operatorname{dom} E}\}$ and $sv = \mathit{Lam}\ x\ e_0\ E$.

  Now, assume sv_1 such that $\vDash sv_1 : \tau_1$. Then, by definition, we also have $\vDash (E, x \mapsto sv_1) : (\Gamma, x : \tau_1)$ and so by the induction hypothesis on $\mathcal{T}_0$ with this, we get $E, x \mapsto sv_1 \vdash e_0 \rightsquigarrow \langle e_0', sv_2 \rangle$ and $\vDash sv_2 : \tau_2$, and a derivation of $[\![E, x \mapsto sv_1]\!]_{\text{tp}} \vdash e_0' : [\![sv_2]\!]_{\text{tp}}$, for some e_0' and sv_2. Thus, by definition of the relation, $\vDash \mathit{Lam}\ x\ e_0\ E : \tau_1 \to \tau_2$.

By definition, $[\![sv]\!]_{\text{tp}} = \{(Lab(y) : [\![sv_y]\!]_{\text{tp}})^{(y \mapsto sv_y) \in E}\}$. For each mapping $(y \mapsto sv_y) \in E$, we have by definition $[\![E]\!]_{\text{tp}}(y) = [\![sv_y]\!]_{\text{tp}}$, and by rule T-Var, we have a derivation $\mathcal{T}_y$ of $[\![E]\!]_{\text{tp}} \vdash y : [\![sv_y]\!]_{\text{tp}}$. Then by T-Rcd on these $\mathcal{T}_y$ derivations, we get the required $[\![E]\!]_{\text{tp}} \vdash e' : [\![sv]\!]_{\text{tp}}$.

- Case $\mathcal{T} = \dfrac{\begin{array}{cc}\mathcal{T}_1 & \mathcal{T}_2\\ \Gamma \vdash e_1 : \tau_2 \to \tau & \Gamma \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash e_1\ e_2 : \tau}$.

  By the induction hypothesis on $\mathcal{T}_1$ with $\mathcal{R}$, we get $\mathcal{D}_1$ of $E \vdash e_1 \rightsquigarrow \langle e_1', sv_1 \rangle$, $\mathcal{R}_1$ of $\vDash sv_1 : \tau_2 \to \tau$, and $\mathcal{T}_1'$ of $[\![E]\!]_{\text{tp}} \vdash e_1' : [\![sv_1]\!]_{\text{tp}}$, for some e_1' and sv_1. Similarly, by the induction hypothesis on $\mathcal{T}_2$, we get $\mathcal{D}_2$ of $E \vdash e_2 \rightsquigarrow \langle e_2', sv_2 \rangle$, $\mathcal{R}_2$ of $\vDash sv_2 : \tau_2$, and $\mathcal{T}_2'$ of $[\![E]\!]_{\text{tp}} \vdash e_2' : [\![sv_2]\!]_{\text{tp}}$, for some e_2' and sv_2.

  By inversion on $\mathcal{R}_1$, $sv_1 = Lam\ x\ e_0\ E_0$, for some x, e_0, and E_0.

  Since $\vDash Lam\ x\ e_0\ E_0 : \tau_2 \to \tau$ and $\vDash sv_2 : \tau_2$, we have by definition of the relation, a derivation $\mathcal{D}_0$ of $E_0, x \mapsto sv_2 \vdash e_0 \rightsquigarrow \langle e_0', sv \rangle$, $\mathcal{R}_0$ of $\vDash sv : \tau$, as required, and $\mathcal{T}_0'$ of $[\![E_0, x \mapsto sv_2]\!]_{\text{tp}} \vdash e_0' : [\![sv]\!]_{\text{tp}}$. By definition, $[\![E_0, x \mapsto sv_2]\!]_{\text{tp}} = [\![E_0]\!]_{\text{tp}}, x : [\![sv_2]\!]_{\text{tp}}$.

  Then, by rule D-App on $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_0$, we get the required:

$$E \vdash e_1\ e_2 \rightsquigarrow \langle \textbf{let } env = e_1' \textbf{ in } (\textbf{ let } y = env.Lab(y) \textbf{ in })^{y \in \text{dom } E_0} \\ \textbf{let } x = e_2' \textbf{ in } e_0', sv \rangle$$

  By Lemma 6 on $\mathcal{T}_1'$, $\mathcal{T}_2'$, and $\mathcal{T}_0'$, we get the required typing derivation.

- Case $\mathcal{T} = \dfrac{\mathcal{T}_0}{\Gamma \vdash e_0 : \{(\ell_i : \tau_i)^{i \in 1..n}\}}\ (1 \le k \le n)$.

  By induction hypothesis on $\mathcal{T}_0$ with $\mathcal{R}$, we get $\mathcal{D}_0$ of $E \vdash e_0 \rightsquigarrow \langle e_0', sv_0 \rangle$, $\mathcal{R}_0$ of $\vDash sv_0 : \{(\ell_i : \tau_i)^{i \in 1..n}\}$, and $\mathcal{T}_0'$ of $[\![E]\!]_{\text{tp}} \vdash e_0' : [\![sv_0]\!]_{\text{tp}}$. By inversion on $\mathcal{R}_0$, $sv_0 = Rcd\ \{(\ell_i \mapsto sv_i)^{i \in 1..n}\}$ and $\vDash sv_i : \tau_i$ by some $\mathcal{R}_i$, for each $i \in 1..n$. By definition, $[\![sv_0]\!]_{\text{tp}} = \{(\ell_i : [\![sv_i]\!]_{\text{tp}})^{i \in 1..n}\}$.

  By rule D-Proj on $\mathcal{D}_0$ with the side condition from $\mathcal{T}$, we get the required $E \vdash e_0.\ell_k \rightsquigarrow \langle e_0'.\ell_k, sv_k \rangle$. We have that $\vDash sv_k : \tau_k$, by $\mathcal{R}_k$. By T-Proj on $\mathcal{T}_0'$ with the side condition from $\mathcal{T}$, we get $[\![E]\!]_{\text{tp}} \vdash e_0'.\ell_k : [\![sv_k]\!]_{\text{tp}}$.

- Case $\mathcal{T} = \dfrac{(\mathcal{T}_i :: \Gamma \vdash e_i : \tau)^{i \in 1..n} \qquad \mathcal{Z} :: \tau\ \text{orderZero}}{\Gamma \vdash [e_1, \ldots, e_n] : []\tau}$.

  For each $i \in 1..n$, by the induction hypothesis on $\mathcal{T}_i$ with $\mathcal{R}$, we get $\mathcal{D}_i$ of $E \vdash e_i \rightsquigarrow \langle e_i', sv_i \rangle$, $\mathcal{R}_i$ of $\vDash sv_i : \tau$, and $\mathcal{T}_i'$ of $[\![E]\!]_{\text{tp}} \vdash e_i' : [\![sv_i]\!]_{\text{tp}}$.

  By pairwise application of Lemma 2 to each of the $\mathcal{R}_i$ together with $\mathcal{Z}$, we get that $sv_1 = \cdots = sv_n$. Let $sv = sv_1$. Then by D-Array on the $\mathcal{D}_i$ derivations, we get $E \vdash [e_1, \ldots, e_n] \rightsquigarrow \langle [e_1', \ldots, e_n'], Arr\ sv \rangle$.

  Since $\vDash sv : \tau$ (by any of the $\mathcal{R}_i$), by Lemma 3 together with $\mathcal{Z}$, we have that $[\![sv]\!]_{\text{tp}} = \tau$. Then by T-Array on the $\mathcal{T}_i'$ derivations and $\mathcal{Z}$, we get that $[\![E]\!]_{\text{tp}} \vdash [e_1', \ldots, e_n'] : []\tau$ and, by definition $[\![Arr\ sv]\!]_{\text{tp}} = [][\![sv]\!]_{\text{tp}}$. Since $\vDash sv : \tau$ and τ orderZero, we get $\vDash Arr\ sv : []\tau$.

- Case $\mathcal{T} = \dfrac{\begin{array}{ccc}\mathcal{T}_0 & \mathcal{T}_1 & \mathcal{T}_2 \\ \Gamma \vdash e_0 : []\tau & \Gamma \vdash e_1 : \mathbf{int} & \Gamma \vdash e_2 : \tau\end{array}}{\Gamma \vdash e_0 \ \mathbf{with}\ [e_1] \leftarrow e_2 : []\tau}$.

  By the induction hypothesis on $\mathcal{T}_0$ with $\mathcal{R}$, we get $\mathcal{D}_0$ of $E \vdash e_0 \rightsquigarrow \langle e_0', sv_0\rangle$, $\mathcal{R}_0$ of $\vDash sv_0 : []\tau$, and $\mathcal{T}_0'$ of $[\![\, E \,]\!]_{\text{tp}} \vdash e_0' : [\![\, sv_0 \,]\!]_{\text{tp}}$. By inversion on $\mathcal{R}_0$, we have that $sv_0 = Arr\ sv_0'$, $\mathcal{R}_0'$ of $\vDash sv_0' : \tau$, and $\mathcal{Z}$ of τ orderZero. By definition, $[\![\, sv_0 \,]\!]_{\text{tp}} = []([\![\, sv_0' \,]\!]_{\text{tp}})$.

  By the induction hypothesis on $\mathcal{T}_1$ with $\mathcal{R}$, we get derivations $\mathcal{D}_1$ of $E \vdash e_1 \rightsquigarrow \langle e_1', sv_1\rangle$, $\mathcal{R}_1$ of $\vDash sv_1 : \mathbf{int}$, and $\mathcal{T}_1'$ of $[\![\, E \,]\!]_{\text{tp}} \vdash e_1' : [\![\, sv_1 \,]\!]_{\text{tp}}$. By inversion on $\mathcal{R}_1$, $sv_1 = Dyn\ \mathbf{int}$, so $[\![\, sv_1 \,]\!]_{\text{tp}} = \mathbf{int}$. By the induction hypothesis on $\mathcal{T}_2$ with $\mathcal{R}$, we get $\mathcal{D}_2$ of $E \vdash e_2 \rightsquigarrow \langle e_2', sv_2\rangle$, $\mathcal{R}_2$ of $\vDash sv_2 : \tau$, and $\mathcal{T}_2'$ of $[\![\, E \,]\!]_{\text{tp}} \vdash e_2' : [\![\, sv_2 \,]\!]_{\text{tp}}$. By Lemma 2 on $\mathcal{R}_0'$, $\mathcal{R}_2$, and $\mathcal{Z}$, we have that $sv_0' = sv_2$. Then by D-Update on $\mathcal{D}_0$, $\mathcal{D}_1$, and $\mathcal{D}_2$, we get:

$$E \vdash e_0 \ \mathbf{with}\ [e_1] \leftarrow e_2 \rightsquigarrow \langle e_0' \ \mathbf{with}\ [e_1'] \leftarrow e_2', Arr\ sv_0'\rangle$$

  Clearly also $[\![\, sv_0' \,]\!]_{\text{tp}} = [\![\, sv_2 \,]\!]_{\text{tp}}$. Then by T-Update on $\mathcal{T}_0'$, $\mathcal{T}_1'$, and $\mathcal{T}_2'$:

$$[\![\, E \,]\!]_{\text{tp}} \vdash e_0' \ \mathbf{with}\ [e_1'] \leftarrow e_2' : []([\![\, sv_0' \,]\!]_{\text{tp}})$$

- Case $\mathcal{T} = \dfrac{\begin{array}{ccc}\mathcal{T}_2 & \mathcal{T}_1 & \mathcal{Z} \\ \Gamma \vdash e_2 : []\tau_2 & \Gamma, x : \tau_2 \vdash e_1 : \tau & \tau\ \text{orderZero}\end{array}}{\Gamma \vdash \mathbf{map}\ (\lambda x.\, e_1)\ e_2 : []\tau}$.

  By the induction hypothesis on $\mathcal{T}_2$ with $\mathcal{R}$, we get $\mathcal{D}_2$ of $E \vdash e_2 \rightsquigarrow \langle e_2', sv_2\rangle$, $\mathcal{R}_2$ of $\vDash sv_2 : []\tau_2$, and $\mathcal{T}_2'$ of $[\![\, E \,]\!]_{\text{tp}} \vdash e_2' : [\![\, sv_2 \,]\!]_{\text{tp}}$, for some e_2' and sv_2. By inversion on $\mathcal{R}_2$, $sv_2 = Arr\ sv_2'$ with $\mathcal{R}_2'$ of $\vDash sv_2' : \tau_2$ (and τ_2 orderZero), and by definition, $[\![\, sv_2 \,]\!]_{\text{tp}} = []([\![\, sv_2' \,]\!]_{\text{tp}})$.

  Using $\mathcal{R}_2'$, we construct $\vDash (E, x \mapsto sv_2') : (\Gamma, x : \tau_2)$. Then by the induction hypothesis on $\mathcal{T}_1$, we get $\mathcal{D}_1$ of $E, x \mapsto sv_2' \vdash e_1 \rightsquigarrow \langle e_1', sv_1\rangle$, $\mathcal{R}_1$ of $\vDash sv_1 : \tau$, and $\mathcal{T}_1'$ of $[\![\, E, x \mapsto sv_2' \,]\!]_{\text{tp}} \vdash e_1' : [\![\, sv_1 \,]\!]_{\text{tp}}$, for some e_1' and sv_1. By definition, $[\![\, E, x \mapsto sv_2' \,]\!]_{\text{tp}} = [\![\, E \,]\!]_{\text{tp}}, x : [\![\, sv_2' \,]\!]_{\text{tp}}$. Then by rule D-Map on $\mathcal{D}_2$ and $\mathcal{D}_1$, we get the required derivation of:

$$E \vdash \mathbf{map}\ (\lambda x.\, e_1)\ e_2 \rightsquigarrow \langle \mathbf{map}\ (\lambda x.\, e_1')\ e_2', Arr\ sv_1\rangle$$

  Since $\vDash sv_1 : \tau$, by $\mathcal{R}_1$, and τ orderZero, by $\mathcal{Z}$, we have $\vDash Arr\ sv_1 : []\tau$, as required. By Lemma 3 on $\mathcal{R}_1$ and $\mathcal{Z}$, we have $[\![\, sv_1 \,]\!]_{\text{tp}} = \tau$. Then by rule T-Map on $\mathcal{T}_2'$, $\mathcal{T}_1'$, and $\mathcal{Z}$, we get a derivation of $[\![\, E \,]\!]_{\text{tp}} \vdash \mathbf{map}\ (\lambda x.e_1')\ e_2' : []\tau$ and we already have that $[\![\, Arr\ sv_1 \,]\!]_{\text{tp}} = []([\![\, sv_1 \,]\!]_{\text{tp}}) = []\tau$.

- The case for T-Loop is similar to the case for T-Map.

□

5.3 Preservation of meaning

In this section, we show that the defunctionalization transformation preserves the meaning of expressions in the following sense: If an expression e evaluates to a value v in an environment Σ, then the translated expression e' will evaluate to

a corresponding value v' in a corresponding environment Σ', and if e evaluates to **err**, then e' will evaluate to **err** in the environment Σ' as well.

The correspondence between values in the source program and target program, and their evaluation environments, will be made precise shortly, but intuitively, we replace each function closure in the source program by a record containing the values in the closure environment.

We first define a simple relation between source language values and static values, given in Figure 5.3, and extend it to relate evaluation environments and defunctionalization environments in the usual way. Note that this relation actually defines a function from values to static values.

$\boxed{\vDash v : sv}$

$$\frac{}{\vDash \overline{n} : \mathit{Dyn}\ \mathbf{int}} \qquad \frac{}{\vDash \mathbf{true} : \mathit{Dyn}\ \mathbf{bool}} \qquad \frac{}{\vDash \mathbf{false} : \mathit{Dyn}\ \mathbf{bool}}$$

$$\frac{\vDash \Sigma : E}{\vDash \mathit{clos}(\lambda x\colon \tau.\ e_0, \Sigma) : \mathit{Lam}\ x\ e_0\ E}$$

$$\frac{(\vDash v_i : sv_i)^{i \in 1..n}}{\vDash \{(\ell_i = v_i)^{i \in 1..n}\} : \mathit{Rcd}\ \{(\ell_i \mapsto sv_i)^{i \in 1..n}\}} \qquad \frac{(\vDash v_i : sv)^{i \in 1..n}}{\vDash [(v_i)^{i \in 1..n}] : \mathit{Arr}\ sv}$$

$\boxed{\vDash \Sigma : E}$

$$\frac{}{\vDash \cdot : \cdot} \qquad \frac{\vDash \Sigma : E \qquad \vDash v : sv}{\vDash (\Sigma, x \mapsto v) : (E, x \mapsto sv)}$$

Figure 5.3: Relation between values and static values, and evaluation environments and defunctionalization environments, respectively.

Next, we define a mapping from source language values to target language values, which simply converts each function closure to a corresponding record expression that contains the converted values from the closure environment:

$$\begin{aligned}
[\![\, v \,]\!]_{\mathrm{val}} &= v\,, \quad \text{for } v \in \{\overline{n}, \mathbf{true}, \mathbf{false}\} \\
[\![\, \mathit{clos}(\lambda x\colon \tau.\ e_0, \Sigma) \,]\!]_{\mathrm{val}} &= \{(\mathit{Lab}(y) = [\![\, v_y \,]\!]_{\mathrm{val}})^{(y \mapsto v_y) \in \Sigma}\} \\
[\![\, \{(\ell_i = v_i)^{i \in 1..n}\} \,]\!]_{\mathrm{val}} &= \{(\ell_i = [\![\, v_i \,]\!]_{\mathrm{val}})^{i \in 1..n}\} \\
[\![\, [(v_i)^{i \in 1..n}] \,]\!]_{\mathrm{val}} &= [([\![\, v_i \,]\!]_{\mathrm{val}})^{i \in 1..n}]
\end{aligned}$$

As we have seen with a number of previous definitions, the case for arrays is actually moot, since arrays will never contain function closures.

We extend this mapping homomorphically to evaluation environments:

$$\begin{aligned}
[\![\, \cdot \,]\!]_{\mathrm{val}} &= \cdot \\
[\![\, \Sigma, x \mapsto v \,]\!]_{\mathrm{val}} &= [\![\, \Sigma \,]\!]_{\mathrm{val}}\,, x \mapsto [\![\, v \,]\!]_{\mathrm{val}}
\end{aligned}$$

The following lemma states that if a value is related to a type of order zero, according to the previously defined relation between values and types (defined in Figure 5.1) used in the proof of type soundness, then the value is mapped to itself, that is, values that do not contain function closures are unaffected by defunctionalization:

Lemma 7. *If* $\vDash v : \tau$ *and* τ orderZero*, then* $[\![v]\!]_{\text{val}} = v$.

Proof. By induction on the derivation of $\vDash v : \tau$. □

Before we can show preservation of meaning, we need a few auxiliary lemmas. The following lemma allows the derivation of an evaluation judgment to be *weakened* by adding unused assumptions, similar to the weakening lemma for typing judgments (Lemma 4).

Lemma 8. *If* $\Sigma \vdash e \downarrow r$ *and* $\Sigma \subseteq \Sigma'$*, then also* $\Sigma' \vdash e \downarrow r$.

Proof. By induction on the structure of the expression e. □

Similar to Lemma 5 and Lemma 6, which are used in the case for application in the proof of Theorem 1, we define analogous lemmas for the evaluation of a translated application.

Lemma 9. *If* $\Sigma, \Sigma_0 \vdash e \downarrow r$*, then*

$$\Sigma, env \mapsto \{(Lab(x) = v_x)^{(x \mapsto v_x) \in \Sigma_0}\} \vdash (\textbf{let } x = env.Lab(x) \textbf{ in})^{x \in \operatorname{dom} \Sigma_0} \ e \downarrow r \ .$$

where env *is a fresh variable not in* $\operatorname{dom} \Sigma, \Sigma_0$.

Proof. By induction on the shape of Σ_0. □

Lemma 10. *If*

$$\begin{gathered} \Sigma \vdash e_1 \downarrow [\![\mathit{clos}(\lambda x \colon \tau.\, e_0, \Sigma_0)]\!]_{\text{val}} , \\ \Sigma \vdash e_2 \downarrow v_2, \ \textit{and} \\ [\![\Sigma_0]\!]_{\text{val}} , x \mapsto v_2 \vdash e_0 \downarrow v \end{gathered}$$

then $\Sigma \vdash \textbf{let } env = e_1 \textbf{ in } (\textbf{let } y = env.Lab(y) \textbf{ in})^{y \in \operatorname{dom} \Sigma_0}$
$\textbf{let } x = e_2 \textbf{ in } e_0 \downarrow v$.

Proof. By a direct proof using Lemma 8 and Lemma 9. □

We are now ready to prove the following theorem, which states that the defunctionalization transformation preserves the meaning of an expression that is known to evaluate to some result, where the value of the defunctionalized expression and the values in the environment are translated according to the translation from source language values to target language values given above.

Theorem 2 (Semantics preservation)**.** *If* $\Sigma \vdash e \downarrow r$ *(by* $\mathcal{E}$*),* $\vDash \Sigma : E$ *(by* $\mathcal{R}$*), and* $E \vdash e \rightsquigarrow \langle e', sv \rangle$ *(by* $\mathcal{D}$*), then if* $r = \textbf{err}$*, then also* $[\![\Sigma]\!]_{\text{val}} \vdash e' \downarrow \textbf{err}$ *and if* $r = v$*, for some value* v*, then* $\vDash v : sv$ *and* $[\![\Sigma]\!]_{\text{val}} \vdash e' \downarrow [\![v]\!]_{\text{val}}$.

Proof. By structural induction on the big-step evaluation derivation $\mathcal{E}$.

We show a few of the more interesting cases:

- Case $\mathcal{E} = \dfrac{}{\Sigma \vdash x \downarrow v}\,(\Sigma(x) = v)$, so $e = x$ and $r = v$.

  $\mathcal{D}$ must be an instance of rule D-Var, so $e' = x$ and $E(x) = sv$. Since $\vDash \Sigma : E$, by assumption $\mathcal{R}$, and $\Sigma(x) = v$, by the side condition, $\mathcal{R}$ must contain a subderivation of $\vDash v : sv$, as required.

  By the side condition from $\mathcal{E}$ and by definition of the mapping on values, $[\![\, \Sigma \,]\!]_{\text{val}}(x) = [\![\, v \,]\!]_{\text{val}}$, so we directly get the required derivation by E-Var:

$$\frac{}{[\![\, \Sigma \,]\!]_{\text{val}} \vdash x \downarrow [\![\, v \,]\!]_{\text{val}}}\,([\![\, \Sigma \,]\!]_{\text{val}}(x) = [\![\, v \,]\!]_{\text{val}})$$

- Case $\mathcal{E} = \dfrac{}{\Sigma \vdash \overline{n} \downarrow \overline{n}}$, so $e = \overline{n}$ and $r = v = \overline{n}$.

  $\mathcal{D}$ must use rule D-Num, so $e' = \overline{n}$ and $sv = \mathit{Dyn}\ \mathbf{int}$. We have the required $\vDash \overline{n} : \mathit{Dyn}\ \mathbf{int}$ by axiom. Since $[\![\, \overline{n} \,]\!]_{\text{val}} = \overline{n}$, we directly get the required evaluation derivation by rule E-Num.

- The cases for E-True and E-False are analogous.

- Case $\mathcal{E} = \dfrac{\mathcal{E}_1 \qquad \mathcal{E}_2}{\Sigma \vdash e_1 \downarrow \mathbf{true} \qquad \Sigma \vdash e_2 \downarrow v}{\Sigma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \downarrow v}$.

  $\mathcal{D}$ must have the following shape:

$$\frac{\mathcal{D}_1 :: E \vdash e_1 \rightsquigarrow \langle e_1', \mathit{Dyn}\ \mathbf{bool}\rangle \qquad \mathcal{D}_2 :: E \vdash e_2 \rightsquigarrow \langle e_2', sv\rangle \qquad \mathcal{D}_3 :: E \vdash e_3 \rightsquigarrow \langle e_3', sv\rangle}{E \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rightsquigarrow \langle \mathbf{if}\ e_1'\ \mathbf{then}\ e_2'\ \mathbf{else}\ e_3', sv\rangle}$$

  By the induction hypothesis on $\mathcal{E}_1$ with $\mathcal{R}$ and $\mathcal{D}_1$, we get derivations $\mathcal{R}_1$ of $\vDash \mathbf{true} : \mathit{Dyn}\ \mathbf{bool}$ and $\mathcal{E}_1'$ of $[\![\, \Sigma \,]\!]_{\text{val}} \vdash e_1' \downarrow [\![\, \mathbf{true} \,]\!]_{\text{val}}$ and by definition $[\![\, \mathbf{true} \,]\!]_{\text{val}} = \mathbf{true}$. Similarly, by the induction hypothesis on $\mathcal{E}_2$ with $\mathcal{D}_2$, we get $\mathcal{R}_2$ of $\vDash v : sv$, as required, and $\mathcal{E}_2'$ of $[\![\, \Sigma \,]\!]_{\text{val}} \vdash e_2' \downarrow [\![\, v \,]\!]_{\text{val}}$.

  Thus, by rule E-IfT on $\mathcal{E}_1'$ and $\mathcal{E}_2'$, we get the required derivation of:

$$[\![\, \Sigma \,]\!]_{\text{val}} \vdash \mathbf{if}\ e_1'\ \mathbf{then}\ e_2'\ \mathbf{else}\ e_3' \downarrow [\![\, v \,]\!]_{\text{val}}$$

- The case for E-IfF is analogous.

- Case $\mathcal{E} = \dfrac{}{\Sigma \vdash \lambda x\colon \tau.\, e_0 \downarrow \mathit{clos}(\lambda x\colon \tau.\, e_0, \Sigma)}$, so $e = \lambda x\colon \tau.\, e_0$.

  $\mathcal{D}$ must be an instance of rule D-Lam, so $e' = \{(\mathit{Lab}(y) = y)^{y \in \operatorname{dom} E}\}$ and $sv = \mathit{Lam}\ x\ e_0\ E$. By assumption $\mathcal{R}$, we have that $\vDash \Sigma : E$, so by definition, $\vDash \mathit{clos}(\lambda x\colon \tau.\, e_0, \Sigma) : \mathit{Lam}\ x\ e_0\ E$, as required.

  By definition, $[\![\, \mathit{clos}(\lambda x\colon \tau.\, e_0, \Sigma) \,]\!]_{\text{val}} = \{(\mathit{Lab}(y) = [\![\, v_y \,]\!]_{\text{val}})^{(y \mapsto v_y) \in \Sigma}\}$, and $\operatorname{dom} E = \operatorname{dom} \Sigma$. For each mapping $(y \mapsto v_y) \in \Sigma$, we have by definition $[\![\, \Sigma \,]\!]_{\text{val}}(y) = [\![\, v_y \,]\!]_{\text{val}}$ and, by E-Var, that $[\![\, \Sigma \,]\!]_{\text{val}} \vdash y \downarrow [\![\, v_y \,]\!]_{\text{val}}$ by some $\mathcal{E}_y$. Thus, by rule E-Rcd on the $\mathcal{E}_y$ derivations, we can construct the required derivation of:

$$[\![\, \Sigma \,]\!]_{\text{val}} \vdash \{(\mathit{Lab}(y) = y)^{y \in \operatorname{dom} \Sigma}\} \downarrow \{(\mathit{Lab}(y) = [\![\, v_y \,]\!]_{\text{val}})^{(y \mapsto v_y) \in \Sigma}\}$$

- Case $\mathcal{E} = \dfrac{\mathcal{E}_1 :: \Sigma \vdash e_1 \downarrow clos(\lambda x\colon \tau.\, e_0, \Sigma_0) \qquad \mathcal{E}_2 :: \Sigma \vdash e_2 \downarrow v_2 \qquad \mathcal{E}_0 :: \Sigma_0, x \mapsto v_2 \vdash e_0 \downarrow v}{\Sigma \vdash e_1\ e_2 \downarrow v}$,

  so $e = e_1\ e_2$ and $r = v$. Then $\mathcal{D}$ must have the following shape:

$$\frac{\mathcal{D}_1 :: E \vdash e_1 \rightsquigarrow \langle e_1', Lam\ x\ e_0\ E_0\rangle \qquad \mathcal{D}_2 :: E \vdash e_2 \rightsquigarrow \langle e_2', sv_2\rangle \qquad \mathcal{D}_0 :: E_0, x \mapsto sv_2 \vdash e_0 \rightsquigarrow \langle e_0', sv\rangle}{E \vdash e_1\ e_2 \rightsquigarrow \langle e', sv\rangle}$$

  where $e' = \mathbf{let}\ env = e_1'\ \mathbf{in}\ (\mathbf{let}\ y = env.Lab(y)\ \mathbf{in})^{y \in \mathrm{dom}\, E_0}\ \mathbf{let}\ x = e_2'\ \mathbf{in}\ e_0'$.

  By the induction hypothesis on $\mathcal{E}_1$ with $\mathcal{R}$ and $\mathcal{D}_1$, we get derivations $\mathcal{R}_1$ of $\vDash clos(\lambda x\colon \tau.\, e_0, \Sigma_0) : Lam\ x\ e_0\ E_0$ and $\mathcal{E}_1'$ of $[\![\, \Sigma \,]\!]_{\mathrm{val}} \vdash e_1' \downarrow [\![\, clos(\lambda x\colon \tau.\, e_0, \Sigma_0) \,]\!]_{\mathrm{val}}$. By inversion on $\mathcal{R}_1$, we get $\mathcal{R}_0$ of $\vDash \Sigma_0 : E_0$ and by definition, $\mathrm{dom}\, \Sigma_0 = \mathrm{dom}\, E_0$. By the induction hypothesis on $\mathcal{E}_2$ with $\mathcal{R}$ and $\mathcal{D}_2$, we get $\mathcal{R}_2$ of $\vDash v_2 : sv_2$ and $\mathcal{E}_2'$ of $[\![\, \Sigma \,]\!]_{\mathrm{val}} \vdash e_2' \downarrow [\![\, v_2 \,]\!]_{\mathrm{val}}$.

  From $\mathcal{R}_0$ and $\mathcal{R}_2$, we can construct $\mathcal{R}_0'$ of $\vDash (\Sigma_0, x \mapsto v_2) : (E_0, x \mapsto sv_2)$. Then, by the induction hypothesis on $\mathcal{E}_0$ with $\mathcal{R}_0'$ and $\mathcal{D}_0$, we get $\vDash v : sv$, as required, and a derivation $\mathcal{E}_0'$ of $[\![\, \Sigma_0, x \mapsto v_2 \,]\!]_{\mathrm{val}} \vdash e_0' \downarrow [\![\, v \,]\!]_{\mathrm{val}}$. By definition, $[\![\, \Sigma_0, x \mapsto v_2 \,]\!]_{\mathrm{val}} = [\![\, \Sigma_0 \,]\!]_{\mathrm{val}}, x \mapsto [\![\, v_2 \,]\!]_{\mathrm{val}}$.

  Finally, by Lemma 10 on $\mathcal{E}_1'$, $\mathcal{E}_2'$, and $\mathcal{E}_0'$, we get the required derivation.

- Case $\mathcal{E} = \dfrac{\mathcal{E}_0 \qquad \mathcal{E}_1}{\Sigma \vdash e_0 \downarrow [(v_i)^{i \in 1..n}] \qquad \Sigma \vdash e_1 \downarrow \overline{k}}\ (k < 1 \vee k > n)$,

  so $e = e_0[e_1]$ and $r = \mathbf{err}$. $\mathcal{D}$ must have the following shape:

$$\frac{\mathcal{D}_0 \qquad \mathcal{D}_1}{E \vdash e_0 \rightsquigarrow \langle e_0', Arr\ sv\rangle \qquad E \vdash e_1 \rightsquigarrow \langle e_1', Dyn\ \mathbf{int}\rangle}{E \vdash e_0[e_1] \rightsquigarrow \langle e_0'[e_1'], sv\rangle}$$

  By the induction hypothesis on $\mathcal{E}_0$ with $\mathcal{R}$ and $\mathcal{D}_0$, we get $\mathcal{E}_0'$ of $[\![\, \Sigma \,]\!]_{\mathrm{val}} \vdash e_0' \downarrow [\![\, [(v_i)^{i \in 1..n}] \,]\!]_{\mathrm{val}}$ and by definition $[\![\, [(v_i)^{i \in 1..n}] \,]\!]_{\mathrm{val}} = [([\![\, v_i \,]\!]_{\mathrm{val}})^{i \in 1..n}]$. By the induction hypothesis on $\mathcal{E}_1$ with $\mathcal{R}$ and $\mathcal{D}_1$, we get $\mathcal{E}_1'$ of $[\![\, \Sigma \,]\!]_{\mathrm{val}} \vdash e_1' \downarrow [\![\, \overline{k} \,]\!]_{\mathrm{val}}$ and by definition $[\![\, \overline{k} \,]\!]_{\mathrm{val}} = \overline{k}$. Then by E-IndexErr on $\mathcal{E}_0'$ and $\mathcal{E}_1'$ with the side condition from $\mathcal{E}$, we get that $[\![\, \Sigma \,]\!]_{\mathrm{val}} \vdash e_0'[e_1'] \downarrow \mathbf{err}$.

- The case for E-UpdateErr is very similar.

- Case $\mathcal{E} = \dfrac{\mathcal{E}_2' :: \Sigma \vdash e_2 \downarrow [(v_i)^{i \in 1..n}] \qquad (\mathcal{E}_i :: \Sigma, x \mapsto v_i \vdash e_1 \downarrow v_i')^{i \in 1..n}}{\Sigma \vdash \mathbf{map}\ (\lambda x.\, e_1)\ e_2 \downarrow [(v_i')^{i \in 1..n}]}$,

  so $e = \mathbf{map}\ (\lambda x.\, e_1)\ e_2$ and $r = v = [(v_i')^{i \in 1..n}]$. $\mathcal{D}$ must be like:

$$\frac{\mathcal{D}_2 \qquad \mathcal{D}_1}{E \vdash e_2 \rightsquigarrow \langle e_2', Arr\ sv_2\rangle \qquad E, x \mapsto sv_2 \vdash e_1 \rightsquigarrow \langle e_1', sv_1\rangle}{E \vdash \mathbf{map}\ (\lambda x.\, e_1)\ e_2 \rightsquigarrow \langle \mathbf{map}\ (\lambda x.\, e_1')\ e_2', Arr\ sv_1\rangle}$$

  By the induction hypothesis on $\mathcal{E}_2'$ with $\mathcal{R}$ and $\mathcal{D}_2$, we get derivations $\mathcal{R}_2$ of $\vDash [(v_i)^{i \in 1..n}] : Arr\ sv_2$ and $\mathcal{E}_2''$ of $[\![\, \Sigma \,]\!]_{\mathrm{val}} \vdash e_2' \downarrow [\![\, [(v_i)^{i \in 1..n}] \,]\!]_{\mathrm{val}}$.

By definition, $[\![\, [(v_i)^{i\in 1..n}] \,]\!]_{\text{val}} = [([\![\, v_i \,]\!]_{\text{val}})^{i\in 1..n}]$. By inversion on $\mathcal{R}_2$, we get $(\mathcal{R}'_i ::\vDash v_i : sv_2)^{i\in 1..n}$.

For each $i \in 1..n$, from $\mathcal{R}$ and $\mathcal{R}'_i$, we construct $\mathcal{R}''_i$ of $\vDash (\Sigma, x \mapsto v_i) : (E, x \mapsto sv_2)$. By the induction hypothesis on $\mathcal{E}_i$ with $\mathcal{R}''_i$ and $\mathcal{D}_1$, we get $\mathcal{R}'''_i$ of $\vDash v'_i : sv_1$ and $\mathcal{E}'''_i$ of $[\![\, \Sigma, x \mapsto v_i \,]\!]_{\text{val}} \vdash e'_1 \downarrow [\![\, v'_i \,]\!]_{\text{val}}$.

From $(\mathcal{R}'''_i)^{i\in 1..n}$, we construct $\vDash [(v'_i)^{i\in 1..n}] : Arr\ sv_1$, as required.

By definition, $[\![\, \Sigma, x \mapsto v_i \,]\!]_{\text{val}} = [\![\, \Sigma \,]\!]_{\text{val}}, x \mapsto [\![\, v_i \,]\!]_{\text{val}}$. Then by E-Map on $\mathcal{E}''_2$ and $(\mathcal{E}'''_i)^{i\in 1..n}$, we get the required:

$$[\![\, \Sigma \,]\!]_{\text{val}} \vdash \textbf{map}\ (\lambda x.\, e'_1)\ e'_2 \downarrow [([\![\, v'_i \,]\!]_{\text{val}})^{i\in 1..n}]$$

- Case $\mathcal{E} = \dfrac{\mathcal{E}_0 :: \Sigma \vdash e_0 \downarrow v_0 \qquad \mathcal{E}_1 :: \Sigma \vdash e_1 \downarrow [(v_i)^{i\in 1..n}] \quad \mathcal{E}\mathcal{L} :: \Sigma; x = v_0; y = (v_i)^{i\in 1..n} \vdash e_2 \downarrow v}{\Sigma \vdash \textbf{loop}\ x = e_0\ \textbf{for}\ y\ \textbf{in}\ e_1\ \textbf{do}\ e_2 \downarrow v}$.

The derivation $\mathcal{D}$ must have the following shape:

$$\frac{\mathcal{D}_0 :: E \vdash e_0 \rightsquigarrow \langle e'_0, sv\rangle \qquad \mathcal{D}_1 :: E \vdash e_1 \rightsquigarrow \langle e'_1, Arr\ sv_1\rangle \quad \mathcal{D}_2 :: E, x \mapsto sv, y \mapsto sv_1 \vdash e_2 \rightsquigarrow \langle e'_2, sv\rangle}{E \vdash \textbf{loop}\ x = e_0\ \textbf{for}\ y\ \textbf{in}\ e_1\ \textbf{do}\ e_2 \rightsquigarrow \langle \textbf{loop}\ x = e'_0\ \textbf{for}\ y\ \textbf{in}\ e'_1\ \textbf{do}\ e'_2, sv\rangle}$$

By the induction hypothesis on $\mathcal{E}_0$ with $\mathcal{R}$ and $\mathcal{D}_0$, we get derivations $\mathcal{R}_0$ of $\vDash v_0 : sv$ and $\mathcal{E}'_0$ of $[\![\, \Sigma \,]\!]_{\text{val}} \vdash e'_0 \downarrow [\![\, v_0 \,]\!]_{\text{val}}$. Again, by the induction hypothesis on $\mathcal{E}_1$ with $\mathcal{R}$ and $\mathcal{D}_1$, we get $\mathcal{R}_1$ of $\vDash [(v_i)^{i\in 1..n}] : Arr\ sv_1$ and $\mathcal{E}'_1$ of $[\![\, \Sigma \,]\!]_{\text{val}} \vdash e'_1 \downarrow [\![\, [(v_i)^{i\in 1..n}] \,]\!]_{\text{val}}$. By inversion on $\mathcal{R}_1$, we get $\vDash v_i : sv_1$ for each $i \in 1..n$. By definition, $[\![\, [(v_i)^{i\in 1..n}] \,]\!]_{\text{val}} = [([\![\, v_i \,]\!]_{\text{val}})^{i\in 1..n}]$.

By an inner induction on the length of a sequence of values $(v'_i)^{i\in 1..n}$, we show that if $\Sigma; x = v'_0; y = (v'_i)^{i\in 1..n} \vdash e_2 \downarrow v'$ (by $\mathcal{E}\mathcal{L}'$), $\vDash v'_0 : sv$ (by $\mathcal{R}'_0$), and $\vDash v'_i : sv_1$ (by $\mathcal{R}'_i$), for each $i \in 1..n$, then $[\![\, \Sigma \,]\!]_{\text{val}}; x = [\![\, v'_0 \,]\!]_{\text{val}}; y = ([\![\, v'_i \,]\!]_{\text{val}})^{i\in 1..n} \vdash e'_2 \downarrow [\![\, v' \,]\!]_{\text{val}}$ and $\vDash v' : sv$.

  - In case the sequence is empty, the derivation $\mathcal{E}\mathcal{L}'$ must be an instance of rule EL-Nil, so in this case we must have that $v' = v'_0$. Then by rule EL-Nil, we get that $[\![\, \Sigma \,]\!]_{\text{val}}; x = [\![\, v'_0 \,]\!]_{\text{val}}; y = \cdot \vdash e'_2 \downarrow [\![\, v'_0 \,]\!]_{\text{val}}$ and by assumption we already have $\vDash v'_0 : sv$.
  - Case $(v'_i)^{i\in 1..n}$. In this case, the derivation $\mathcal{E}\mathcal{L}'$ must be like:

$$\frac{\mathcal{E}_2 \qquad\qquad \mathcal{E}\mathcal{L}''}{\Sigma, x \mapsto v'_0, y \mapsto v'_1 \vdash e_2 \downarrow v''_0 \qquad \Sigma; x = v''_0; y = (v'_i)^{i\in 2..n} \vdash e_2 \downarrow v'}{\Sigma; x = v'_0; y = (v'_i)^{i\in 1..n} \vdash e_2 \downarrow v'}$$

    From $\mathcal{R}$, $\mathcal{R}'_0$, and $\mathcal{R}'_1$, we construct a derivation of $\vDash (\Sigma, x \mapsto v'_0, y \mapsto v'_1) : (E, x \mapsto sv, y \mapsto sv_1)$. By the outer induction hypothesis on $\mathcal{E}_2$ with this relation and $\mathcal{D}_2$, we get $\mathcal{R}_2$ of $\vDash v''_0 : sv$ and $\mathcal{E}'_2$ of $[\![\, \Sigma, x \mapsto v'_0, y \mapsto v'_1 \,]\!]_{\text{val}} \vdash e'_2 \downarrow [\![\, v''_0 \,]\!]_{\text{val}}$ and by definition, $[\![\, \Sigma, x \mapsto v'_0, y \mapsto v'_1 \,]\!]_{\text{val}} = [\![\, \Sigma \,]\!]_{\text{val}}, x \mapsto [\![\, v'_0 \,]\!]_{\text{val}}, y \mapsto [\![\, v'_1 \,]\!]_{\text{val}}$. Then by the inner induction hypothesis on $\mathcal{E}\mathcal{L}''$ with $\mathcal{R}_2$ and $(\mathcal{R}'_i)^{i\in 2..n}$,

we get $\mathcal{EL}'''$ of

$$[\![\,\Sigma\,]\!]_{\text{val}}\,; x = [\![\,v_0''\,]\!]_{\text{val}}\,; y = ([\![\,v_i'\,]\!]_{\text{val}})^{i\in 2..n} \vdash e_2' \downarrow [\![\,v'\,]\!]_{\text{val}}$$

and $\vDash v' : sv$, as required. By EL-Cons on $\mathcal{E}_2'$ and $\mathcal{EL}'''$, we get:

$$[\![\,\Sigma\,]\!]_{\text{val}}\,; x = [\![\,v_0'\,]\!]_{\text{val}}\,; y = ([\![\,v_i'\,]\!]_{\text{val}})^{i\in 1..n} \vdash e_2' \downarrow [\![\,v'\,]\!]_{\text{val}}$$

Taking v_i' to be v_i, for $i \in 0..n$, and v' to be v, in the above argument, we get the required $\vDash v : sv$ and a derivation $\mathcal{EL}_0$ of:

$$[\![\,\Sigma\,]\!]_{\text{val}}\,; x = [\![\,v_0\,]\!]_{\text{val}}\,; y = ([\![\,v_i\,]\!]_{\text{val}})^{i\in 1..n} \vdash e_2' \downarrow [\![\,v'\,]\!]_{\text{val}}$$

Then by rule E-Loop on $\mathcal{E}_0'$, $\mathcal{E}_1'$, and $\mathcal{EL}_0$, we get the required derivation.

- Case $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Sigma \vdash e_1 \downarrow \mathbf{err}\end{array}}{\Sigma \vdash e_1 + e_2 \downarrow \mathbf{err}}$, so $e = e_1 + e_2$ and $r = \mathbf{err}$.

  The derivation $\mathcal{D}$ must use rule D-Plus and contain a subderivation $\mathcal{D}_1$ of $E \vdash e_1 \rightsquigarrow \langle e_1', \mathit{Dyn}\ \mathbf{int}\rangle$. By the induction hypothesis on $\mathcal{E}_1$ with $\mathcal{R}$ and $\mathcal{D}_1$, we get $\mathcal{E}_1'$ of $[\![\,\Sigma\,]\!]_{\text{val}} \vdash e_1' \downarrow \mathbf{err}$. Then by rule E-PlusErr1 on $\mathcal{E}_1'$, we get the required derivation of $[\![\,\Sigma\,]\!]_{\text{val}} \vdash e_1' + e_2' \downarrow \mathbf{err}$.

- The other cases that propagate **err** are similar.

□

Note that we did not assume the source expression to be well-typed. As mentioned previously, this is because the translation rules inherently perform some degree of type checking. For example, an expression like

$$\mathbf{if}\ b\ \mathbf{then}\ (\lambda x\colon \mathbf{int}.\ x + n)\ \mathbf{else}\ (\lambda x\colon \mathbf{int}.\ x + m)$$

is not well-typed since the branches have order 1, but it will not translate to anything either, since the constraint in the rule D-If, that the static value of each branch must be identical, cannot be satisfied.

5.4 Correctness of defunctionalization

To summarize the previous properties and results relating to the correctness of the defunctionalization transformation, we state the following corollary which follows by type soundness (Lemma 1), normalization and preservation of typing for defunctionalization (Theorem 1), and semantics preservation of defunctionalization (Theorem 2), together with Lemma 3 and Lemma 7.

Corollary 1 (Correctness)**.** *If* $\vdash e : \tau$ *and* τ orderZero*, then* $\vdash e \downarrow r$*, for some* r*,* $\vdash e \rightsquigarrow \langle e', sv\rangle$*, for some* e' *and* sv*, and* $\vdash e' : \tau$ *and* $\vdash e' \downarrow r$ *as well.*

Chapter 6

Implementation

The defunctionalization transformation that was presented in Chapter 4 has been implemented in the Futhark compiler, which is developed in the open on GitHub and publicly available at `https://github.com/diku-dk/futhark`.

The primary contribution of the implementation work of this thesis is the defunctionalization compiler pass, which was implemented from scratch. Additionally, a monomorphization pass was also added to the compiler pipeline.[1] The type checker was extended to work for higher-order programs and refined with the type restrictions that we have presented. Furthermore, the type checker was also generalized to better handle the instantiation of polymorphic functions, which served as a starting point for the implementation of type inference, although this extension was not implemented by the author.

In this chapter, we discuss various aspects of the implementation of defunctionalization in Futhark. We discuss how our implementation diverges from the theoretical description and present a few optimizations that have been made. As Futhark is a real language with a fairly large number of syntactical constructs, as well as features such as uniqueness types for supporting in-place updates and size-dependent types for reasoning about the sizes of arrays, it would not be feasible to do a formal treatment of the entire language.

Futhark supports a small number of parallel higher-order functions, such as `map`, `reduce`, `scan`, and `filter`, as compiler intrinsics. These are specially recognized by the compiler and exploited to perform optimizations and generate parallel code. User-defined parallel higher-order functions are ultimately defined in terms of these. As a result, the program produced by the defunctionalizer is not *exclusively* first-order, but may contain fully saturated applications of these built-in higher-order functions.

6.1 Overview

In the Futhark compiler, defunctionalization is implemented as a whole-program, source-to-source transformation. Specifically, defunctionalization takes an entire monomorphic, module-free source language program as input and produces another well-typed, first-order Futhark program as output. Note that the in-

[1] These passes are implemented in the `Futhark.Internalise.Defunctionalise` module and in the `Futhark.Internalise.Monomorphise` module, respectively.

put must be monomorphic. We will explain this, and the interaction between defunctionalization and polymorphism, in more detail in Section 6.2.

The relevant parts of the compilation pipeline are as follows. First, after lexical analysis and parsing, the source program is type checked. Second, static interpretation of the higher-order module language is performed to yield a complete Futhark program without any modules. Third, polymorphic functions are specialized to concrete types to yield a completely monomorphic program and type synonyms are removed. Fourth, the module-free, monomorphic Futhark program is defunctionalized. After this, the first-order, monomorphic, module-free Futhark program is transformed to an intermediate language representation and the remainder of the compilation back end proceeds.

6.2 Polymorphism and defunctionalization

Futhark supports parametric polymorphism in the form of let-polymorphism (or ML-style polymorphism). The defunctionalizer, however, only works on monomorphic programs and therefore, programs are *monomorphized* before being passed to the defunctionalizer. To achieve this, a monomorphization pass has been implemented. This pass simply records all polymorphic functions occurring in a given program and specializes each of them for each distinct set of type instantiations, as determined by the applications occurring in the program. The type checker has been extended to attach the instantiated types to every variable, which makes monomorphization a fairly simple procedure.

An alternative approach would have been to perform defunctionalization directly on polymorphic programs, but since the monomorphization itself is relatively simple, the current approach was chosen to allow the defunctionalization transformation to be as simple as possible.

Since the use of function types should be restricted as described earlier, it is necessary to distinguish between type variables which may be instantiated with function types, or any type of order greater than zero, and type variables which may only take on types of order zero. Without such distinction, one could write an invalid program that we would not be able to defunctionalize, for example by instantiating the type variable `a` with a function type in the following:

```
let ite 'a (b: bool) (x: a) (y: a) : a =
  if b then x else y
```

To prevent this from happening, we have introduced the notion of *lifted type variables*, written `'^a`, which are unrestricted in the types that they may be instantiated with, while the regular type variables may only take on types of order zero. Consequently, a lifted type variable must be considered to be of order greater than zero and is thus restricted in the same way as function types. The regular type variables, on the other hand, can be treated as having order zero and may be used freely in conditionals, arrays, and loops.

6.3 Optimizations

In the theoretical presentation of defunctionalization, the translation rule for application, D-App, will insert a copy of the translated body of the applied

function at every application site. Effectively, this means that all functions will be inlined, which is also evident from the fact that function applications are actually completely eliminated by the translation. Inlining all functions can produce very large programs if the same functions are called in multiple locations. In the implementation, we instead perform a variant of lambda lifting [21], where a new top-level function is created, which takes as arguments the record corresponding to the closure environment, that is, the defunctionalized applied expression, and the original defunctionalized argument of the application. This lifted function accesses the values of the environment (via a record pattern in the environment parameter) and has the translated body of the function that was originally applied as its body. The original application is then replaced by a call to this function.

The basic defunctionalization algorithm only considers functions that take a single argument. For example, if a curried function f of type $\tau_1 \to \tau_2 \to \tau_3$ is applied to arguments e_1 and e_2 as $f\ e_1\ e_2$, then this will be treated as a partial application $f\ e_1$, followed by another application of this partially applied function to e_2. This simple approach results in the creation of many trivial lifted functions, which simply take an environment and a single argument and repackages them in a new environment. For each partial application, a new function will be created. Only the function that accepts the final argument to fully saturate the original function will perform anything interesting. For fully-applied calls to first-order functions, in particular, this is completely unnecessary and will cause the translation of existing first-order programs to significantly increase the size of the programs for no reason.

To remedy this excess creation of trivial lifted functions, we have extended the notion of static values in the implementation with a *dynamic function*:

$$sv ::= \cdots \mid \mathit{DynFun}\ \langle e, sv \rangle\ sv$$

A dynamic function can be viewed as a functional analogue to a dynamic value. In particular, the application of a dynamic function is allowed to remain in the defunctionalized program, although partial application of dynamic functions needs to be handled specially, as will be detailed shortly.

A *DynFun* is essentially a list of closures, where each element $\langle e, sv \rangle$ consists of an expression e representing the closure environment and a static value sv for the code part of the closure. Element i in the list (counting from 0) corresponds to the closure obtained by partially applying the dynamic function to its first i arguments. The final static value of the list is the static value for the result of fully applying the dynamic function.

To illustrate this idea, consider the following function:

```
let f (x: i32) (y: i32) (g: i32->i32) : i32 =
  g x + y
```

Even though this function is not first-order, we can still transform it, as follows, to allow applications of the function to its first two arguments to remain in the transformed program:

```
let f (x: i32) (y: i32) = {x=x, y=y}
```

This function will be represented by a static value of the following form:

$$\begin{aligned}&\mathit{DynFun}\ \langle\{\}, \mathit{Lam}\ x\ (\lambda y\colon \mathbf{i32}.\ \lambda g\colon \mathbf{i32} \to \mathbf{i32}.\ g\ x + y)\ \cdot\rangle\\ &\quad(\mathit{DynFun}\ \langle\{x = x\}, \mathit{Lam}\ y\ (\lambda g\colon \mathbf{i32} \to \mathbf{i32}.\ g\ x + y)\ (\cdot, x \mapsto \mathit{Dyn}\ \mathbf{i32})\rangle\\ &\qquad(\mathit{Lam}\ g\ (g\ x + y)\ (\cdot, x \mapsto \mathit{Dyn}\ \mathbf{i32}, y \mapsto \mathit{Dyn}\ \mathbf{i32})))\end{aligned}$$

When transforming a function application, we keep track of the depth of application. If we eventually find that the applied expression is a variable that denotes a dynamic function, then if the function is fully applied, that is, if the depth of application corresponds to the length of the *DynFun* list, then the application is left in place and only the arguments are transformed. The static value at the end of the *DynFun* list becomes the static value for the entire application. However, if the dynamic function is only partially applied with i arguments, then we create a new function with the first i parameters and the expression at position i of the *DynFun* static value as body. Then the applied function variable is replaced with the name of the new function and the static value of the entire partial application becomes the static value at position i.

For instance, the application `f 1` will translate to `f' 1`, where `f'` is a new lifted function of the form

```
let f' (x: i32) = {x=x}
```

and the static value of the application becomes the following:

$$\mathit{Lam}\ y\ (\lambda g\colon \mathbf{i32} \to \mathbf{i32}.\ g\ x + y)\ (\cdot, x \mapsto \mathit{Dyn}\ \mathbf{i32})$$

Despite these improvements, defunctionalization may still produce a large number of rather trivial lifted functions. The optimizations performed later in the compilation pipeline will eliminate all of these trivial functions (we will discuss this in more detail in Chapter 7), but this can significantly increase compilation times. To further reduce the number of trivial functions produced, the defunctionalizer immediately inlines certain simple functions, in particular those that just contain a record expression or a single primitive operation.

6.4 Array shape parameters

Futhark employs a system of runtime-checked size-dependent types, where array types have the symbolic sizes of their dimensions attached to them. The programmer may give shape declarations in function definitions to express invariants about the shapes of arrays that a function takes as arguments or returns as result, and these shapes may be used as term-level variables as well. Shape annotations may also be used in type ascriptions.

If a function that expresses an invariant about the shapes of its array parameters is partially applied or contains another parameter of function type, the defunctionalizer will create a new function that captures the array argument in an environment record, which would subsequently be passed as argument to another function that receives the next argument, and so on. This causes the array arguments to be separated and could potentially destroy the shape invariant. For example, consider partially applying a function such as the following (where `[n]` denotes a shape parameter):

```
let f [n] (xs: [n]i32) (ys: [n]i32) = ...
```

In the implementation, we preserve the connection between the shapes of the two array parameters by capturing the shape parameter `n` along with the array parameter `xs` in the record for the closure environment and then extend the Futhark internalizer to handle dependency between value parameters and shapes of array parameters, and insert a dynamic check to ensure that they are equal. In the case of the function `f`, the defunctionalized program will look something like the following:

```
let f^ {n: i32, xs: []i32} (ys: [n]i32) = ...
let f [n] (xs: [n]i32) = {n=n, xs=xs}
```

The Futhark compiler will then insert a dynamic check to verify that the size of array `ys` is equal to the value of argument `n`.

Of course, built-in operations that truly rely on these invariants, such as `zip`, will perform this shape check regardless, but by maintaining these invariants in general, we prevent code from silently breaching the contract that was specified by the programmer through the shape annotations in the types.

Having extended Futhark with higher-order functions, it might be useful to also be able to specify shape invariants on function type parameters and function type results, and on expressions of function type in general. This is not supported by the existing type system and in general it would not possible in the current type system to verify a shape constraint on an expression of function type. We can still ensure that a shape invariant is checked dynamically, upon applying a function, by repeatedly eta-expanding the function expression and inserting type ascriptions with shape annotations on the order-zero parameters and bodies. For instance, the following type ascription,

```
e : ([n]i32 -> [m]i32) -> [m]i32
```

would be translated as follows:

```
\x -> (e (\(y:[n]i32) -> x y : [m]i32)) : [m]i32
```

This extension has not yet been implemented in Futhark.

Shape parameters are not explicitly passed on application, but rather they are implicitly inferred from the arguments of the value parameters. When function types are allowed to refer to these shapes, it must be ensured that each shape is given in positive position at least once before it is used in negative position, that is, each shape may not be used in a function type parameter until it has been given in a parameter of order zero type at least once. This refinement has been implemented in the Futhark type checker.

6.5 In-place updates

In this section, we describe the interaction between in-place updates and higher-order functions, the complications that arise, and the simple but conservative restriction that was implemented to maintain soundness. For the record, this solution was devised and implemented by Troels Henriksen.

Futhark supports in-place modification of arrays using a type system extension based on *uniqueness types* to ensure safety and referential transparency [18]. An array may be consumed by being the source of an in-place update, e.g.,

a **with** $[i] \leftarrow e$, or by being passed as argument for a unique function parameter (indicated by an asterisk on the parameter type).

The type of an expression indicates its uniqueness attributes. For example, the following type characterizes a function which consumes its first argument:

```
*[]a -> b -> c
```

Unfortunately, this type does not accurately describe *when* the array argument is consumed. It could be consumed immediately, in case the function first performs an in-place update and then returns another function, or it could not be consumed until all arguments have been passed to the function.

This ambiguity was not present in first-order Futhark since functions could not return other functions and functions could not be partially applied. However, in higher-order Futhark, a function may be partially applied and this makes the safety analysis of in-place updates significantly more complicated.

Consider a function `f` of the above type. If this function is partially applied to an array `xs`, it is not clear from the type whether `xs` should be considered consumed at that point, and thus prohibited from being used again, or whether `xs` is not consumed until `f` receives its second argument. Furthermore, if we bind the partial application `f xs` to a variable `g`, it is not clear if it is safe to apply `g` multiple times or not.

Thus, to ensure safety we have to assume that an array argument is consumed immediately when passed for a unique parameter, that is, `xs` cannot be used after the partial application `f xs`. We also have to assume that a partially applied function, which may consume its argument, will perform an in-place update upon further application, that is, `g` may only be applied once. The restriction that a partially applied consuming function may only be applied once is rather tricky to enforce, in particular if the function is passed as argument to another function.

These complications have been resolved by fairly simple, but very conservative restrictions in the type checker: First, if the bound expression in a **let**-binding has a function type (or any type of order greater than zero), then this expression may not perform any in-place updates. Second, a partially applied function, that consumes one of its arguments, may not be passed as argument to another function.

Alternatively, the type system could be extended to make use of effect types to capture when the effect of performing an in-place update can take place. However, this would make the language more complicated to use.

In conclusion, higher-order functions and in-place updates do not interact in a modular manner and there does not seem to be a straightforward way to improve this situation, aside from introducing more sophisticated type analyses.

Chapter 7

Empirical evaluation

We may consider two different aspects in the empirical evaluation of our implementation of restricted higher-order functions in Futhark through defunctionalization. The first aspect is whether defunctionalization yields efficient programs, or in other words, whether the use of higher-order functions carries an overhead at run-time. The second aspect is whether the restricted higher-order functions are actually useful and whether they allow the programmer to reap some of the benefits that we claimed in the introduction, such as increased modularity, or whether the restrictions are too severe in practice.

7.1 Performance evaluation

It is somewhat difficult to evaluate the first aspect in a rigorous way, since it relies on having actual higher-order programs that can reasonably be compared to equivalent first-order programs, but translating first-order programs to use higher-order functions is a manual process and there are no objective criteria for judging the right amount and right use of higher-order functions. Certainly, the transformation leaves the performance of existing programs completely unchanged, since the implemented transformation only affects programs that use first-class functions or higher-order functions, as a result of the optimizations described in Section 6.3.

We can, however, report on some of the changes to the existing Futhark code that the addition of higher-order functions have inspired and how these changes have affected performance. A particularly interesting effect is the significant rewriting of the Futhark standard library, Futlib, that the support for higher-order functions has enabled. Most of the Futhark SOACs have been converted to library functions that wrap compiler intrinsics, including `map`, `reduce`, `scan`, `filter` and others. Furthermore, various higher-order utility functions have been added, such as function composition and application, `curry`, and `uncurry`. Sorting functions, parameterized over some comparison operator, have been translated from using parametric modules to using higher-order functions, and similarly, a segmented scan operation now uses higher-order functions.

Futhark has a fairly extensive suite of benchmark programs translated from the benchmark suites of Accelerate [5], Rodinia [7], Parboil [30] and others. All of these programs make heavy use of library functions, in particular SOACs, since they are the means by which parallelism is expressed. Thus, the per-

formance impact of using higher-order functions has been indirectly evaluated through the use of the standard library in the benchmark programs. In addition to the use of library-defined SOACs, most benchmark programs have been rewritten to make use of the various higher-order utility functions where appropriate. The use of higher-order functions did not affect the run-time performance of the benchmark programs. Thus, our restricted higher-order functions may be considered a *zero-cost abstraction* and Futhark programmers need not worry about any negative consequences of extensive use of higher-order functions in their code.

This positive result relies on the extensive optimizations performed by the Futhark compiler, since defunctionalization still results in a fairly convoluted program, despite the optimizations that were made to the implementation of defunctionalization, as described in the previous chapter. The most important of these optimizations are inlining, copy propagation, and hoisting of let-expressions out of parallel SOACs and loops.

7.2 Programming with restricted higher-order functions

We now consider the second aspect, namely evaluating the usefulness of our restricted higher-order functions. This is perhaps the more interesting one, since the answer may not be a simple affirmative. Indeed, certain programs simply cannot be written because of these restrictions, without completely changing the basic algorithms of these programs. In the following, we will show that various important styles of functional programs are unaffected by our type restrictions. We will also give some negative examples and discuss what particular kinds of programs we cannot write.

Functional images

The representation of data as functions is fundamental to λ-calculus, with canonical examples such as Church encoding [9] of natural numbers, Booleans, and lists. Other more modern examples include the representation of environments by their lookup functions and the representation of arrays by their indexing functions, such as the *delayed arrays* utilized in Repa [23]. Another interesting use case is the idea of *functional images* [12], where an image is represented by a function from points on the plane to colors, or any kind of "pixel" representation. In Futhark, we can represent this as follows:

```
type point     = (f32, f32)
type image 'a = point -> a
```

Image manipulation and transformation is then defined via higher-order functions or simply as function composition. Additionally, animations can be defined as images abstracted over time, for some suitable type `time`:

```
type anim 'a = time -> image a
```

Pan [12, 13] is a domain specific language (EDSL) embedded in Haskell that implements this idea. Values in Pan, such as the floating point numbers that make up the points in the domain of images, are actually program fragments that represent these numbers. Thus, a Haskell meta-program written using Pan actually generates another program and the Pan library embeds a compiler

into the meta-program which performs various optimizations on the generated program at run-time and compiles it to C code.

The entire Pan library has been translated to Futhark. This work was done by Troels Henriksen and Martin Elsman (the supervisors of this thesis). Interestingly, this translation was very straightforward and the type restrictions of higher-order Futhark did not demand any adjustments to be made compared to the Haskell implementation. This is likely because of the staged compilation approach that Pan uses, which requires that functions be restricted in ways that are essentially identical to our restrictions, so that the compiled Haskell program can generate an efficient C program.

To illustrate how defunctionalization works on functional images, we now give a relatively simple example of a functional image and a higher-order function that works as an image filter to transform this image. The source program is shown in Figure 7.1a and the program that results from defunctionalization and simplification is shown in Figure 7.1b. This example is taken from [12]. The functional image is simply a vertical line through the origin, defined by the function `vstrip`. The image filter `swirl r` is given as a function on generic images and it performs a kind of "swirling" transformation on images with any codomain (any pixel representation). An example of a rendered image produced by this program is given in Figure 7.2.

The actual program produced by the defunctionalizer is much more complicated. In particular, it contains a lot of packing and unpacking of environment records containing nothing or little more than the values for global variables like `pi` or the empty closure environments of library functions like `cos` and `sin`. We have eliminated these for clarity and also replaced the internal variable names generated by the compiler. The functions `sqrt`, `abs`, `cos`, and so on, in the translated program, are compiler intrinsic functions (with simplified names). On the other hand, the program that is produced after passing through the entire compiler pipeline is extremely simple with everything inlined, but it hardly resembles the original program anymore and so it does not illustrate defunctionalization so well.

In the Futhark implementation, as opposed to the original Pan EDSL, high-performance GPU code is directly generated at compile time. After defunctionalization and various compiler optimizations have been performed on a functional image program, the resulting program is essentially just a simple two-dimensional map, that computes the color value of each pixel in the raster image representation in parallel.

Dynamic nesting of closures

Defunctionalization is a whole-program transformation and it relies on the fact that every function abstraction is present in the program text. In particular, the code part of every closure is available in the source program even if the number of closures at run-time may be dynamic, because each abstraction may be evaluated in different environments any number of times. In itself, the fact that the number of closures can be dynamic is not a problem for our defunctionalization algorithm. Consider, for example, the following contrived Futhark program, which instantiates the single λ-abstraction with as many different environments as the length of the input array:

```
type point     = (f32, f32)
type img 'a    = point -> a
type transform = point -> point
type filter 'a = img a -> img a

let vstrip : img bool =
  \(x,_) -> f32.abs x <= 0.5f32

let dist0 ((x,y): point) : f32 = f32.sqrt(x*x+y*y)

let rotateP (theta: f32) : transform =
  \(x,y) -> (x * f32.cos theta - y * f32.sin theta,
             y * f32.cos theta + x * f32.sin theta)

let swirlP (r: f32) : transform =
  \p -> rotateP (dist0 p * 2 * f32.pi / r) p

let swirl 'a (r: f32) : filter a =
  \im -> im <<| swirlP (-r)

let image : img bool = \p -> swirl 1 vstrip p
```

(a) Source program.

```
let dist0 ((x,y): (f32,f32)) : f32 = sqrt (x*x+y*y)

let swirl (r: f32) (p: (f32,f32)) : (f32,f32) =
  let theta = (dist0 p * 2 * pi) / r
  let ((x,y) : (f32,f32)) = p
  in ((x * cos theta) - (y * sin theta),
      (y * cos theta) + (x * sin theta))

let swirl_vstrip (r: f32) (p: (f32,f32)) : bool =
  let ((x,_): (f32,f32)) = swirl r p
  in abs x <= 0.5f32

let image (p: (f32,f32)) : bool =
  swirl_vstrip (-1) p
```

(b) Resulting first-order program.

Figure 7.1: Example of a program implementing a functional image and a "swirl" image filter using higher-order functions, and the resulting first-order program produced by defunctionalization and simplification.

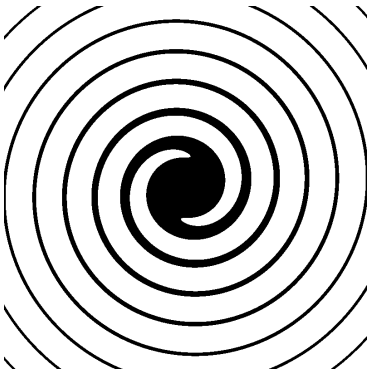

Figure 7.2: Swirl filter applied to a functional image of a vertical line. This image was rendered using the example program given in Figure 7.1a.

```
let main (xs: []i32) =
  loop y = 0 for x in xs do (\z -> z+x) y
```

After defunctionalization, the λ-abstraction will be replaced by a record `{x=x}`, which will be passed to a function that extracts `x` and evaluates `z+x`.

Closures may be arbitrarily nested, that is, the environment of a closure may contain closures, which may themselves also contain closures and so on. In general, nested closures may be constructed through recursion or any other kind of looping control structure, and so the depth of nesting can depend on dynamic information. Our defunctionalization method can only handle nested closures of a statically known depth. This constraint is ensured by the restriction on functions in loops, enforced by the T-Loop typing rule. To see how the dynamic nesting of closures causes problems, consider the following Futhark program:

```
let main (xs: []i32) (y: i32) =
  let g = loop f = (\z -> z + 1) for x in xs
          do (\z -> x + f z)
  in g y
```

The problem is that the function `g` that results from the loop may be of the form `(\z -> z + 1)` or it may be of the form `(\z -> x + f z)`, for some function `f` that is itself of either one of those two forms. This suggests a recursive data type representation such as

$$\mu\,\alpha.\ \{\} + \{x\colon \mathbf{i32}, f\colon \alpha\}\,,$$

which is the kind of representation that would be used in Reynolds-style defunctionalization of the above program, like the example in Section 2.1.

If the variable `f` did not occur in the λ-abstraction in the body of the loop, this program would not be very problematic, since the code part of the closure produced by the loop would be statically known to be either one of the two options (depending on whether the loop performed zero or more iterations) and the environment part would be produced by the loop at run-time.

Sets as characteristic functions

We now consider the very simple example, adapted from [25], of representing sets by their characteristic functions, in order to investigate the impact of the restrictions on higher-order functions in a simple setting.

In Futhark, we can represent generic sets and set insertion (specialized to integers to avoid having to pass an equality operator) as follows:

```
type set 'a = a -> bool
let empty_set 'a : set a = \_ -> false
let insert (x: i32) (s: set i32) : set i32 =
  \y -> (x == y) || s y
```

Because of the type restrictions, it would seem that any function that produces a set as a result is restricted to a control flow that could be expressed completely in the form of "straight-line code", that is, without any control flow that depends on dynamic information and thus could be entirely inlined. Fundamentally, this is true because this is exactly what the type restrictions attain, namely that any function can be statically determined. For example, if we have a program that uses `insert` a given number of times to build up a set of some statically known size (which is, in a sense, the only thing that is possible under our restrictions), then defunctionalization will yield a number of specialized lookup functions for different sizes of sets, with sets being represented as nested records of some fixed depth, quite like a linked list encoded using records. An example of these "chained" lookup functions is shown in Figure 7.3.

This phenomenon is similar to the observation made by Reynolds [26], that a functional environment turns into a linked list when subjected to defunctionalization. Although in our case, the data representation is not recursive, but simply encoded statically as nested records. Hence, our version of defunctionalization produces a chain of calls to different specialized lookup functions rather than having a single recursive function that traverses the list.

```
let lookup1 {x = (x: i32),
             s = (s: {}) } (y: i32) : bool =
  (x == y) || (let (_: i32) = y in false)

let lookup2 {x = (x: i32),
             s = (s: {x: i32,
                      s: {}})} (y: i32) : bool =
  (x == y) || lookup1 s y

let lookup3 {x = (x: i32),
             s = (s: {x: i32,
                      s: {x: i32,
                          s: {} }})} (y: i32) : bool =
  (x == y) || lookup2 s y
```

Figure 7.3: Program resulting from defunctionalization of sets as characteristic functions and a number of set insertions.

This restriction to straight-line code, however, does not mean that dynamic control flow and first-class functions are completely incompatible. For example, we can encode a conditional that returns a set as follows:

```
let if_set 'a (b: bool) (s1: set a)
                       (s2: set a) : set a =
  \x -> if b then s1 x else s2 x
```

This is just the eta-expanded version of the immediate, but illegal implementation. In a similar way, we can work around the type restrictions and write a function that folds an array into a set:

```
let fold_set (xs: []i32) : set i32 =
  \y -> loop found = false
        for x in xs do found || x == y
```

(This could be implemented more efficiently using a reduction or a while-loop.)

These two examples illustrate how the type restrictions can often be worked around. In common for both functions is that they essentially just *delay* the dynamic decision that is inevitable in the computation. Essentially, we simply cannot fold an array into a characteristic function, but we can embed the iteration over the array into the definition of the characteristic function itself.

We note, however, that we cannot in general support functions in conditionals simply by eta-expansion, as we will discuss further in the next chapter.

Segmented scan

A *segmented scan* [2] is a generalization of the usual scan, which performs the scan operation independently on specified segments of the input array. We can derive segmented scan from the ordinary scan: From a given associative, binary operator and identity element, we can derive a new associative operator and corresponding identity, which performs the segmented scan when used in a normal scan over the input array together with an array of flags indicating the segments. Among other things, segmented scans can be used to express some computations on irregular data structures.

In first-order Futhark, `scan` is a built-in SOAC, but without support for user-defined higher-order functions, it was not possible to write a general segmented scan function. Previously, this limitation was circumvented by using the higher-order module system instead, to express the instantiation of a segmented scan for a particular operator and identity element as the application of a parametric module to a module representation of a monoid. This works reasonably well, but it requires some amount of boilerplate code and the segmented scan operation cannot be instantiated "on the fly", but needs to defined as a new module on the top level for each particular operation.

The original implementation of segmented scan from Futlib, using higher-order modules, is shown in Figure 7.4a (modified slightly). We have translated this to use higher-order functions instead, as shown in Figure 7.4b.

These two implementations are very similar on the surface, but the first one is expressed in terms of a strongly normalizing, simply typed λ-calculus built on top of Futhark, in the form of a module system, while the second version is expressed directly in higher-order Futhark. Both are specialized to

```
module type monoid = { type t
                       val ne : t
                       val op : t -> t -> t }

module segmented_scan(M: monoid): {
  val segmented_scan : []bool -> []M.t -> []M.t
} = {
  let segmented_scan [n] (flags: [n]bool)
                         (as: [n]M.t) : [n]M.t =
    (unzip (scan (\(x_flag,x) (y_flag,y) ->
                  (x_flag || y_flag,
                   if y_flag then y else M.op x y))
           (false, M.ne)
           (zip flags as))).2
}
```

(a) Using higher-order modules.

```
let segmented_scan [n] 't (op: t -> t -> t) (ne: t)
                          (flags: [n]bool)
                          (as: [n]t) : [n]t =
  (unzip (scan (\(x_flag,x) (y_flag,y) ->
                (x_flag || y_flag,
                 if y_flag then y else op x y))
         (false, ne)
         (zip flags as))).2
```

(b) Using higher-order functions.

Figure 7.4: Two different implementations of segmented scan in Futhark.

first-order Futhark code at compile time, through static interpretation and defunctionalization, respectively.

The biggest difference for the programmer is how they are used. The version using higher-order functions can be used completely as if it was an ordinary SOAC like `scan`, that is, it may be used anywhere for any given operator and identity element, like

```
segmented_scan (*) 1 flags xs
```

whereas the module version requires that a monoid module be defined and a `segmented_scan` parametric module be instantiated, as in the following:

```
module i32mult = {
  type t = i32
  let ne = 1
  let op x y = x * y
}
module s = segmented_scan(i32mult)
```

```
let main (flags: []bool) (xs: []i32) =
  s.segmented_scan flags xs
```

Furthermore, the form of the operator for the module argument of the `segmented_scan` parametric module is restricted to the form given in the module parameter type. In particular, the operator cannot depend on some additional argument, without making this explicit in the parametric module, and its implementation can only refer to other top-level definitions. Using higher-order functions, on the other hand, the operator can be the result of a function composition or computed by any other function type expression and it may refer to values in a local scope.

In the above example, the programs that result from static interpretation of modules and defunctionalization of higher-order functions, respectively, are quite similar in their basic structure, although the code produced for the module version is very clean and readable compared to the code produced by defunctionalization, which includes a lot of the usual unnecessary repackaging of environments and so on.

This difference is likely explained by the fact that higher-order modules are inherently more restricted than our higher-order functions, and the static interpretation of modules is essentially just the usual λ-calculus evaluation of the module language yielding terms of the underlying Futhark language, whereas defunctionalization needs to handle partial application, closures escaping their scope, and others. However, the programs that result from running the optimizing Futhark compilation pipeline on the two programs in Figure 7.4 are completely identical, up to renaming of bound variables.

Chapter 8

Extensions

8.1 Support for function-type conditionals

Given that the main novelty enabling efficient defunctionalization is the restrictions in the type system, it is interesting to consider how these restrictions could be loosened to allow more programs to be typed and transformed, and what consequences this would have for the efficiency of the transformed programs.

In the following, we consider lifting the type restriction on conditionals, so that branches may produce functions of any order. This change introduces a binary choice for the static value of a conditional and this choice may depend on dynamic information. The inferred static value must capture this choice. Thus, we may extend the definition of static values as follows:

$$sv ::= \cdots \mid Or\ sv_1\ sv_2$$

It is important not to introduce more branching than necessary, so the static values of the branches of a conditional should be appropriately combined to isolate the dynamic choice at much as possible. In particular, if a conditional returns a record, the *Or* static value should only be introduced for those record fields that produce *Lam* static values.

The residual expression for a functional value occurring in a branch must be extended to include some kind of token to indicate which branch is taken at run time, so that the translated program can dynamically determine which function to apply. Unfortunately, it is fairly complicated to devise a translation that preserves typeability in the current type system. The residual expression of a function occurring in a nested conditional would need to include as many tokens as the maximum depth of nesting in the outermost conditional. Additionally, the record capturing the free variables in a function would need to include the union of all the free variables in each λ-abstraction that can be returned from that conditional. Hence, we would have to include "dummy" record fields for those variables that are not in scope in a given function, and "dummy" tokens for functions that are not deeply nested in branches.

What is needed to remedy this situation, is the addition of (binary) sum types to the language:

$$\tau ::= \cdots \mid \tau_1 + \tau_2$$

If we add binary sums, along with expression forms for injections and case-

```
let r = if c1
        then (if c2 then {f = \x -> x+k, a = 3}
                    else {f = \x -> x+n, a = 7})
        else {f = \x -> x+k+n, a = 42}
in r.f r.a
```

(a) Source expression.

```
let r = if c1
        then (if c2
              then {f = inl (inl {k=k}), a = 3}
              else {f = inl (inr {n=n}), a = 7})
        else {f = inr {k=k, n=n}, a = 42}
in let x = r.a
   in case r.f of
        inl s -> (case s of
                    inl e -> (let k = e.k in x+k)
                    inr e -> (let n = e.n in x+n)
        inr e -> (let k = e.k
                  let n = e.n in x+k+n)
```

(b) Target expression.

Figure 8.1: Example of defunctionalization of a nested conditional expression that returns a result containing a functional value.

matching, the transformation would just need to keep track of which branches were taken to reach a particular function-type result and then wrap the usual residual expression in appropriate injections. An application of an expression with an *Or* static value would then perform pattern matching until it reaches a *Lam* static value and then insert **let**-bindings to put the closed-over variables into scope, for that particular function.

To illustrate this extended transformation, consider the Futhark expression in Figure 8.1a, where `k` and `n` are some integer variables in scope, and `c1` and `c2` are expressions of type **bool**. By isolating the dynamic choice, as described above, the static value for the expression `r` would have the following structure (with some details omitted for the sake of brevity):

$$\begin{aligned} &\mathit{Rcd}\ \{f \mapsto \mathit{Or}\ \ (\mathit{Or}\ (\mathit{Lam}\ \ldots)\ (\mathit{Lam}\ \ldots))\ \ (\mathit{Lam}\ \ldots),\\ &\quad a \mapsto \mathit{Dyn}\ \mathbf{int}\} \end{aligned}$$

Using sum types to represent the dynamic choice between functions, we can translate this expression to the expression given in Figure 8.1b. In this extended language, `inl` and `inr` are left and right injections into a binary sum, while the case expression matches these two options.

An interesting aspect of this extension is that it appears to very strictly delimit the set of possible functions that may occur at a given call site, which is an attractive feature for GPU code generation in particular. Rather than by performing general Reynolds-style defunctionalization and then using types

or control-flow analysis to make the apply function well-typed and limit the amount of control-flow, we can instead only introduce branching as necessary.

8.2 Support for while-loops and recursion

To increase the expressive power of the language presented in Chapter 3, we might consider adding a form of while-loop, as also present in Futhark, in addition to the bounded loop over the elements of an array:

$$e ::= \cdots \mid \textbf{loop}\ x = e_0\ \textbf{while}\ e_1\ \textbf{do}\ e_2$$

As with our existing loops, clearly we cannot allow while-loops that produce functional values. If this restriction is made, then the addition of while-loops does not cause any problems for defunctionalization and we can still prove that defunctionalization terminates and preserves typing. However, the language would no longer be strongly normalizing, so the current formulation of type soundness in Lemma 1 would no longer hold. Consequently, Corollary 1 would not hold either. Thus, we would need to reformulate those two properties to assume, rather than conclude, the derivability of the evaluation judgment

Another extension that we might consider is the support for recursion and how that would interact with defunctionalization. For the current project, this extension is mostly of theoretical interest since GPUs offer only very limited support for stacks, although Futhark could potentially be extended to allow only (mutually) recursive functions that are tail-recursive.

Based on preliminary investigations, it seems very likely that our defunctionalization method can support mutually recursive first-order functions without any significant problems. The main interesting aspect of this is the fact that recursive functions may partially apply themselves or pass themselves as arguments to other functions (which may not be recursive under these restrictions, since they would not be first-order). A partially applied recursive call would be handled in a very similar way as how partial application of "dynamic" first-order functions is currently handled, except that the defunctionalization environment would include the static value for the function itself. It would also require some more bookkeeping to ensure that lifted functions, containing recursive calls, are translated to call themselves recursively rather than producing more identical functions ad infinitum.

In general, we cannot allow recursive functions that return functions, since that may cause the creation of dynamically nested closures, as discussed in relation to functions in loops in Section 7.2. Consider, for example, the following contrived function:

```
let f (x: int) : int -> int =
  if x < 0 then \y -> y
           else \y -> x + f (x-1) y
```

We cannot possibly determine the form of the function returned by `f`.

Similarly, we cannot in general allow recursive functions that take function arguments, since a recursive function may construct an arbitrarily nested closure in the function argument in recursive calls. As a simple example, consider the following function:

```
let f (g: int -> int) (x: int) : int =
  if x < 0 then g x
           else f (\y -> g y + g y) (x-1)
```

Similar to the previous example, the function that is applied in the final recursive call of `f` cannot possibly be statically determined, since the parameter `g` accumulates a nested function closure with a shape that we cannot determine until run-time.

However, we may be able to support recursive functions with function parameters as long as the function parameters do not accumulate as in the previous example. This may be rather similar to the variable-only criterion of Chin and Darlington [8], which they introduce to avoid non-termination when specializing recursive functions to their functional arguments.

Going further: Unrestricted dynamic nesting of closures

As discussed in Section 7.2, what makes it difficult to handle functions in loops is the construction of dynamically nested closures, which prevents us from statically determining the shape of the function produced by a loop. The same is true for functions constructed through recursion. As we mentioned, this problem suggests the addition of sum types and recursive types.

We envision how we could potentially support unrestricted functions in loops, and potentially recursive functions, by extending the type system with recursive sum types and introducing recursive static values together with the *Or* static value, which is essentially a *sum* static value.

Recall the example, given in Section 7.2, of a loop that produces a nested closure of a statically unknown depth. We repeat it here again for convenience:

```
let main (xs: []i32) (y: i32) =
  let g = loop f = (\z -> z + 1) for x in xs
          do (\z -> x + f z)
  in g y
```

If we extend the translation with recursive static values, we could represent the expression bound to `g` with a static value of the following form:

$$\mu\,\alpha.\ \mathit{Or}\ (\mathit{Lam}\ z\ (z+1)\ \cdot)\\ \qquad (\mathit{Lam}\ z\ (z+fz)\ (\cdot, x \mapsto \mathit{Dyn}\ \mathbf{int}, f \mapsto \alpha))$$

The loop could be translated to construct a nested record of the recursive type that we gave in Section 7.2 and the application of `g` could be translated to an application of a lifted function `g'`, of the following form, to `g` and the original argument:

```
let g' (env: mu T. {} + {x: i32, f: T}) (z: i32) =
  case env of
    inl env' -> z + 1
    inr env' -> let x = env'.x
                let f = env'.f in x + g' f z
```

Here `mu T. U` is a recursive type, which binds the type variable `T` in type `U`.

We note that this function is rather similar to the apply function of Reynolds-style defunctionalization, except that this function is specialized to the application of the function produced by the single loop. Clearly, we are approaching a topic that is not particularly relevant to GPU compilation since the result of a transformation like this would likely be very inefficient on GPUs, if it could even work in the general case. However, it is still rather interesting if this method could be used to devise a more general, type-preserving, Reynolds-style defunctionalization algorithm that introduces only a minimal amount of branching, without relying on subsequent type-based specialization of apply functions or a separate control-flow analysis. We leave this investigation for future work.

Chapter 9

Conclusion

In this concluding chapter of the thesis, we discuss some related work, mention a few potential directions for future work, and finally conclude.

9.1 Related work

We already mentioned the original work by Reynolds in Section 2.1, where we showed a simple example and pointed out the issue with typing the transformed program. In a simply-typed setting, in order for a defunctionalized program to be well-typed, the apply function needs to be specialized to each particular type of function in the original program. Further complications arise when using Reynolds-style defunctionalization in a polymorphic setting; the apply function may receive a piece of data representing a polymorphic function, which would require the apply function to accept different argument types, depending on the particular form of the encoded function argument. Bell et al. [1] describe a type-driven defunctionalization method that resolves this issue by specializing each apply function to the type of their function-representing arguments, and specializing the functions that call the apply functions, as necessary. Thus, they only perform monomorphization as necessary during defunctionalization, which may avoid excessive code duplication when possible. For our purposes, we are not so concerned about this particular issue, since most function calls in Futhark will be inlined anyway, for execution on the GPU.

Pottier and Gauthier [25] take a different approach and manage to prove type preservation for defunctionalization with a single apply function for System F extended with guarded algebraic data types. They also point out that the specialization of apply functions to each different function type, and even specialization to different number of arguments, can result in many highly specialized apply functions, each having only a small number of branches. This is an interesting perspective in the context of GPU code generation, although one may imagine that many practical programs use first-class functions of only a limited set of types.

A significant amount of research has been made in the area of embedded domain specific languages (EDSLs) for data-parallel programming, including languages such as Accelerate [5] and Obsidian [31], both of which target GPUs.

Accelerate is embedded in Haskell and uses a *staged compilation* approach, where the compiled Accelerate programs generate and compile CUDA GPU

programs at run-time. Since Accelerate programs are really Haskell programs that are written using the Accelerate library, the meta-programs may use the full power of Haskell, including unrestricted higher-order functions. However, the generated programs themselves are first-order and the embedding of the Accelerate language prevents arrays from containing functions and it prevents the construction of GPU computations that produce functions. Accelerate effectively disallows for functions to be represented on the GPU, except for the CUDA kernels themselves, but because of staging, these restrictions are worked around by using the facilities of Haskell and delaying the GPU compilation.

Obsidian is similar in that it uses a staged approach and thus allows the use of higher-order functions in the meta-language. As a GPU language, Obsidian offers more fine-grained control over the details of constructing GPU kernels, whereas Accelerate offers a higher level of abstraction, but is restricted to a set of hand-tuned algorithmic skeletons.

Data Parallel Haskell (DPH) [6] follows in the footsteps of the seminal work by Blelloch on NESL [3]. NESL was targeted at a vector execution model with limitations resembling those of modern GPUs and it does not support higher-order functions. DPH extends the nested data-parallel programming model of NESL to the full Haskell languages and DPH does support higher-order functions using closure conversion, however, the compilation target of DPH is multi-core CPUs rather than massively parallel processors like GPUs.

The GPU language Harlan [19] is remarkable for its wide range of features. Harlan supports first-class functions by using Reynolds-style defunctionalization, which uses the support for algebraic data types that Harlan also has, to represent the function closures. Since this approach inherently involves branching and representation of irregularly sized data, Harlan will most likely suffer from the performance issues that we have worked to avoid by completely eliminating first-class functions at compile-time. The authors of Harlan also note these performance concerns, but state that it has not caused problems yet. However, most of the Harlan benchmarks do not make much use of closures on the GPU. The authors also note that some of these problems could be mitigated by using control flow analysis to delimit the set of functions that can occur at each application site.

NOVA [10] is similar to Harlan in that it is an independent data-parallel language, that can compile to GPU code and supports sum types and higher-order functions. However, it is not very clear from the description in the paper how closures are implemented on the GPU.

Single Assignment C (SaC) [29] is a parallel functional array language focused on numerical computations. SaC does not support higher-order functions due to concerns about their effect on performance, specifically the creation of closures [28]. As demonstrated in this thesis, these concerns can be eliminated by typing rules that guarantee the possibility of efficient defunctionalization.

The basic proof technique of logical relations for proving strong normalization for the simply-typed λ-calculus is originally due to Tait [32] and later generalized to System F by Girard [16]. This technique has been the inspiration for our approach to proving termination and preservation of typing for defunctionalization in this thesis.

Minamide et al. [24] describe a type-directed and type-preserving closure conversion transformation for the simply-typed λ-calculus and System F and prove it correct using a logical relations argument. This is conventional clo-

sure conversion and their work is not concerned with data parallelism or the implementation of closures without relying on function pointers. They also discuss the issue encountered when typing a transformed conditional expression that returns functions with different sets of free variables, as mentioned in Section 8.1. They resolve this issue by representing closures as packages of existential type that abstract the type of the environment. As pointed out by Danvy and Nielsen [11], what defunctionalization attains is exactly the representation of this existential type by a finite sum type and corresponding injections and case dispatch.

Defunctionalization, in the style of Reynolds, has been used as an important implementation strategy in a number of works [4, 33]. For instance, the Standard ML compiler MLton [4] uses defunctionalization, together with a control-flow analysis, to implement higher-order functions, and it performs most optimizations on the first-order intermediate representation. Like Futhark, MLton also performs defunctorization and monomorphization before defunctionalization. Unlike our method, MLton inserts dispatches over functions at call-sites, although their control-flow analysis limits the number of cases required and the authors show that the cost of dispatches is small in practice. Of course, MLton targets CPUs where branching is not much of an issue.

Partial evaluation [22] is a very general approach to program optimization and particularly program specialization. While we are not directly specializing programs to some statically known inputs, we can still view our defunctionalization transformation from the perspective of partial evaluation, which has been inspirational for our work. In analogy to binding-time analysis and the static and dynamic values of partial evaluation, we consider all functional values to be static and we propagate information about functions throughout the program in order to perform specialization of static information. Unlike general program specialization, our type restrictions ensure that all functions remain statically known and that none will be residualized.

9.2 Future work

We have already discussed a couple of ideas for future work in Chapter 8. The most interesting of these is probably the support for function-type conditionals and since there is already plans for adding sum types to Futhark, this would certainly be a reasonable task to pursue once that is done.

In Section 6.4, we mentioned how we could support shape invariants on function type parameters and results, and on expressions of function type. This feature still remains to be added to Futhark.

Various other improvements could be made to the defunctionalization algorithm, as implemented in Futhark, to improve the resulting program. However, the compiler will almost always optimize away any of the unnecessary repackaging of environments, duplicated and unnecessary lifted functions, and so on. Thus, the optimizations made to the defunctionalizer should not be overly intricate, since they would mostly just serve to decrease compilation times and perhaps make the defunctionalized programs more readable for a human, rather than improve the actual performance of compiled code.

Similar to dynamic functions, the handling of higher-order functions that take multiple arguments could be improved so that the lifted function takes

as many arguments as given, rather than creating a new function for every argument. This could likely be done in an elegant way, which would not significantly complicate the translation. The static value *Lam* for functions is currently somewhat "weak" in its representation, in the sense that the expression for the function body is just an unconstrained expression, even when we know that a function takes multiple arguments. The static value representation could be refined to more accurately characterize the shapes of the functions that they represent. Similarly, the *DynFun* representation is also weak in the representation of closures for partially applied functions. It may be possible to unify these two notions to make the system more elegant.

Another direction of future work is to improve the interaction between higher-order functions and in-place updates, which is currently very restricted and not really modular. It may be, however, that the design of the uniqueness type system needs to be rethought in the context of a higher-order language, or that a clean, modular solution is not really feasible without sacrificing programming convenience.

9.3 Closing remarks

In this thesis, we have shown a useful approach to implementing higher-order functions in high-performance functional languages for restrictive compilation targets like GPUs. This approach uses a defunctionalization transformation that exploits type-based restrictions on the use of functions to remove all higher-order functions without introducing any branching into the resulting first-order program. We have proven the correctness of this transformation. Furthermore, we have successfully implemented this transformation in the Futhark compiler and we have described the extensions and optimizations that were made in the implementation. Lastly, we have evaluated the implementation and have found that the use of higher-order functions does not add any overhead, and that the restricted higher-order functions are indeed a practical and useful addition to the language.

Bibliography

[1] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 25–37, New York, NY, USA, 1997. ACM.

[2] Guy E. Blelloch. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[3] Guy E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM (CACM)*, 39(3):85–97, 1996.

[4] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Gert Smolka, editor, *European Symposium on Programming*, pages 56–71. Springer, 2000.

[5] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[6] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, 10 2009.

[8] Wei-Ngan Chin and John Darlington. A higher-order removal method. *Lisp and Symbolic Computation*, 9(4):287–322, 1996.

[9] Alonzo Church. *The calculi of lambda-conversion.* Number 6 in Annals of Mathematics Studies. Princeton University Press, 1941.

[10] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. NOVA: A Functional Language for Data Parallelism. In *Procs. of Int. Workshop on Libraries, Languages, and Compilers for Array Prog.*, ARRAY'14, pages 8:8–8:13, New York, NY, USA, 2014. ACM.

[11] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 162–174. ACM, 2001.

[12] Conal Elliott. Functional images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003.

[13] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003. Updated version of paper by the same name that appeared in SAIG '00 proceedings.

[14] Martin Elsman. Static interpretation of modules. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, pages 208–219, New York, NY, USA, 1999. ACM.

[15] Martin Elsman, Troels Henriksen, Danil Annenkov, and Cosmin Oancea. Static interpretation of higher-order modules in Futhark. In *Proceedings of the 2018 ACM SIGPLAN International Conference on Functional Programming*, ICFP'18. ACM, 2018. To appear.

[16] Jean Yves Girard. Interpretation Fonctionnelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur. In *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[17] Troels Henriksen. *Design and Implementation of the Futhark Programming Language*. PhD thesis, Department of Computer Science, Faculty of Science, University of Copenhagen, 2017.

[18] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 556–571. ACM, 2017.

[19] Eric Holk, Ryan Newton, Jeremy Siek, and Andrew Lumsdaine. Region-based memory management for GPU programming languages: Enabling rich data structures on a spartan host. *ACM SIGPLAN Notices*, 49(10):141–155, 2014.

[20] John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, February 1989.

[21] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[22] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[23] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM.

[24] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283. ACM, 1996.

[25] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 89–98, New York, NY, USA, 2004. ACM.

[26] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

[27] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[28] Sven-Bodo Scholz. Single Assignment C - functional programming using imperative style. In *In John Glauert (Ed.): Proceedings of the 6th International Workshop on the Implementation of Functional Languages. University of East Anglia*, 1994.

[29] Sven-Bodo Scholz. Single Assignment C: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[30] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[31] Joel Svensson, Mary Sheeran, and Koen Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *Symposium on Implementation and Application of Functional Languages*, pages 156–173. Springer, 2008.

[32] William W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

[33] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.