High-Performance Defunctionalisation in Futhark

Anders Kiel Hovgaard [0000−0001−7166−1613]
Troels Henriksen [0000−0002−1195−9722]
Martin Elsman ✉ [0000−0002−6061−5993]

DIKU, University of Copenhagen, Denmark
hzs554@alumni.ku.dk athas@di.ku.dk mael@di.ku.dk

Abstract. General-purpose massively parallel processors, such as GPUs, have become common, but are difficult to program. Pure functional programming can be a solution, as it guarantees referential transparency, and provides useful combinators for expressing data-parallel computations. Unfortunately, higher-order functions cannot be efficiently implemented on GPUs by the usual means. In this paper, we present a defunctionalisation transformation that relies on type-based restrictions on the use of expressions of functional type, such that we can 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 run-time performance, and we argue that we gain many of the benefits of general higher-order functions, without in most practical cases being hindered by the restrictions.

Keywords: Defunctionalisation · GPGPU · Compilers.

1 Introduction

Higher-order functional languages enable programmers to write abstract, compositional, and modular programs [24], and are often considered well-suited for parallel programming, due to the lack of shared state and side effects. The emergence of commodity massively parallel processors, such as GPUs, has exacerbated the need for developing practical techniques for programming parallel hardware. However, GPU programming is notoriously difficult, since GPUs offer a significantly more restricted programming model than that of CPUs. For example, GPUs do not readily allow for higher-order functions to be implemented, mainly because GPUs have only limited support for function pointers.

If higher-order functions cannot be implemented directly, we may opt to remove them by means of a program transformation that replaces them by a simpler language mechanism. The canonical such transformation is *defunctionalisation*, which was first described by Reynolds [31]. Reynolds' defunctionalisation abstracts each functional value by a set of records representing each particular instance of the function, and the functional values in a program are abstracted by the disjoint union of these sets. Each application in a program is then replaced

```
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

Fig. 1: Example demonstrating the defunctionalisation transformation

by a call to an *apply* function, which performs a case match on each of the functional forms and essentially serves as an interpreter for the functional values in the original program. The most basic form 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 threads in a GPU execute together in lockstep, in so called *warps* of usually 32 threads, a large amount of branching will cause many threads to be idle in the branches where they are not executing instructions.

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. These restrictions allow defunctionalisation by specializing each application to the particular form of function that may occur at run time. The result is essentially equivalent to inlining completely the *apply* function in a program produced by Reynolds defunctionalisation. Notably, the transformation does not introduce any additional branching.

We have used the Futhark language [17,18,19,20,21] 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, we show a simple Futhark program in Figure 1a and the resulting program after defunctionalisation in Figure 1b (simplified slightly). The result is a first-order program that explicitly pass closure environments, in the form of records capturing the free variables, in place of first-class functions in the source program.

The principal contributions of this paper are:

– A defunctionalisation transformation expressed on a simple data-parallel functional array language, with type rules that restrict the use of higher-

order functions to allow for the defunctionalisation to remove effectively higher-order functions in all cases, without introducing any branching.
- 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.
- A description and evaluation of the transformation as implemented in the compiler for a real high-performance functional language (Futhark).

In the following, we use the notation $(\mathcal{Z}_i)^{i \in 1..n}$ 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, and so on. Further, we 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.

2 Language

To be able to formally define and reason about the defunctionalisation transformation, to be presented in Section 3, 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, operational semantics, and type system.

2.1 Syntax

The set of *types* of the source language is given by the following grammar. The meta-variable $\ell \in \mathbf{Lab}$ 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 meta-variable $x \in \mathbf{Var}$ 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 $n \in \mathbb{Z}$.

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

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 meta theory.

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, $\tau\,\mathrm{orderZero}$, given by the following rules, which assert that a type τ does not contain any function type as a subterm:

$$\frac{}{\mathbf{int}\,\mathrm{orderZero}} \qquad \frac{}{\mathbf{bool}\,\mathrm{orderZero}} \qquad \frac{(\tau_i\,\mathrm{orderZero})^{i\in 1..n}}{\{(\ell_i\colon \tau_i)^{i\in 1..n}\}\,\mathrm{orderZero}} \qquad \frac{\tau\,\mathrm{orderZero}}{[\,]\tau\,\mathrm{orderZero}}$$

2.2 Typing Rules

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 $\tau\,\mathrm{orderZero}$ 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.

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 by a typing context is denoted by $\mathrm{dom}\,\Gamma$ 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 $x \in \mathrm{dom}\,\Gamma$. The typing rules for the language are given in Figure 2.

2.3 Semantics

For the sake of the meta theory presented later, we choose to define a big-step operational semantics with an evaluation environment and function closures.

[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.

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

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

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

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

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

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

$$\text{T-Let: } \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau} \qquad \text{T-Rcd: } \frac{(\Gamma \vdash e_i : \tau_i)^{i \in 1..n}}{\Gamma \vdash \{(\ell_i = e_i)^{i \in 1..n}\} : \{(\ell_i : \tau_i)^{i \in 1..n}\}}$$

$$\text{T-Proj: } \frac{\Gamma \vdash e_0 : \{(\ell_i : \tau_i)^{i \in 1..n}\}}{\Gamma \vdash e_0.\ell_k : \tau_k}\ (1 \leq k \leq n) \qquad \text{T-Length: } \frac{\Gamma \vdash e_0 : [\,]\tau}{\Gamma \vdash \mathbf{length}\ e_0 : \mathbf{int}}$$

$$\text{T-Array: } \frac{(\Gamma \vdash e_i : \tau)^{i \in 1..n} \qquad \tau\ \mathrm{orderZero}}{\Gamma \vdash [e_1, \ldots, e_n] : [\,]\tau} \qquad \text{T-Index: } \frac{\Gamma \vdash e_0 : [\,]\tau \qquad \Gamma \vdash e_1 : \mathbf{int}}{\Gamma \vdash e_0[e_1] : \tau}$$

$$\text{T-Update: } \frac{\Gamma \vdash e_0 : [\,]\tau \quad \Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_0\ \mathbf{with}\ [e_1] \leftarrow e_2 : [\,]\tau} \qquad \text{T-Map: } \frac{\tau\ \mathrm{orderZero} \quad \Gamma \vdash e_2 : [\,]\tau_2 \qquad \Gamma, x : \tau_2 \vdash e_1 : \tau}{\Gamma \vdash \mathbf{map}\ (\lambda x.\ e_1)\ e_2 : [\,]\tau}$$

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

Fig. 2: Typing rules

Evaluation environments Σ and *values* v are defined mutually inductively:

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

Evaluation environments Σ map variables to values and have the same properties and notations as the typing context with regards to extension, variable lookup, and distinctness of variables. A function *closure*, denoted $\mathit{clos}(\lambda x\colon \tau.\ e_0, \Sigma)$, is a value that captures the environment in which a λ-abstraction was evaluated. The values of the target language are the same, but without function closures.

Because the language involves array indexing and updating that may fail, 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**.

The big-step operational semantics for the language is given by the derivation rules in Figure 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. Techniques exist for limiting this duplication [6,30], but, for simplicity, we have chosen a traditional style of presentation.

The rule E-Loop refers to an auxiliary judgment form, defined in Figure 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.

3 Defunctionalisation

We now define the defunctionalisation 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.

Translation environments (or *defunctionalisation environments*) E and *static values* sv are defined mutually inductively, as follows:

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

Translation environments map variables to static values. We assume the same properties as we did for typing contexts and evaluation environments, and we use analogous notation. As the name suggests, a static value is essentially a static approximation of the value that an expression will eventually evaluate to. Static values resemble 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*, which approximates functional values, will contain the actual function parameter and body, along with a defunctionalisation environment containing static values approximating the values in the closed-over environment. The two other constructors *Rcd* and *Arr* complete the correspondence between types and static values.

The defunctionalisation translation takes place in a defunctionalisation 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3: Big-step operational semantics

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.

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

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

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

Fig. 4: Auxiliary judgment for the semantics of loops

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 5 and Figure 6.

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

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

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

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

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

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

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

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

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

Fig. 5: Derivation rules for the defunctionalisation transformation

In the implementation, the record in the residual expression of rule D-Lam captures only the free variables in the λ-abstraction. Likewise, the defunctionalisation environment embedded in the static value is restricted to the free vari-

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

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

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

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

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

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

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

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

$$\text{D-Loop:}\ \frac{\begin{array}{c} E \vdash e_1 \rightsquigarrow \langle e_1', sv \rangle \qquad E \vdash e_2 \rightsquigarrow \langle e_2', \mathit{Arr}\ sv_2 \rangle \\ E, x \mapsto sv, y \mapsto sv_2 \vdash e_3 \rightsquigarrow \langle e_3', sv \rangle \end{array}}{\begin{array}{l} E \vdash \mathbf{loop}\ x = e_1\ \mathbf{for}\ y\ \mathbf{in}\ e_2\ \mathbf{do}\ e_3 \\ \qquad \rightsquigarrow \langle \mathbf{loop}\ x = e_1'\ \mathbf{for}\ y\ \mathbf{in}\ e_2'\ \mathbf{do}\ e_3', sv \rangle \end{array}}$$

Fig. 6: Derivation rules for the defunctionalisation transformation (cont.)

ables. This refinement is not hard to formalise, 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. Interestingly, this constraint on the static values of branches 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 translates any order-zero expression into an equivalent expression that does not contain any higher-order functions. Any first-order expression can be translated by converting the types of its parameters (which are necessarily order zero) to static values, by mapping record types to *Rcd* static values and base types to *Dyn* static values, and including these as bindings for the parameter variables in an initial translation environment.

By a relatively simple extension to the system, we can 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.

4 Meta Theory

In this section, we show type soundness and argue for the correctness of the defunctionalisation transformation presented in Section 3. 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 defunctionalised expression is equivalent to the meaning of the original expression.

4.1 Type Soundness and Normalisation

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 also establish that the language is strongly normalizing, which comes as no surprise given the lack of recursion and bounded number of iterations of loops.

To this end, we first define a relation between values and types, given by derivation rules in Figure 7, and extend it to relate evaluation environments and typing contexts.

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

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

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

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

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

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

Fig. 7: Relation between values and types, and evaluation environments and typing contexts, respectively

We then state and prove type soundness as follows. We do not go into the details of the proof and how the relation between values and types is used. The cases for T-Lam and T-App are the most interesting in this regard, but we omit the details in favor of other results which more directly pertain to defunctionalisation. A similar relation and its role in the proof of termination and preservation of typing for the defunctionalisation transformation is described in more detail in Section 4.2.

Lemma 1 (Type Soundness). *If $\Gamma \vdash e : \tau$ (by $\mathcal{T}$) and $\vDash \Sigma : \Gamma$, 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 the case for T-Lam, we prove the implication in the premise of the rule relating closure values and function types. In the case for T-App, we use this implication to obtain the needed derivations for the body of the closure. In the case for T-Loop, in the subcase where the first two subexpressions evaluate to values, we proceed by an inner induction on the structure of the corresponding sequence of values for the loop iterations. □

4.2 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, with a typing context and type that can be obtained from the defunctionalisation environment and the static value, respectively.

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:

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

This mapping is extended to map defunctionalisation 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 defunctionalisation, 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 defunctionalisation environments and typing contexts. This relation is given by the rules in Figure 8.

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

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

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

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

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

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

Fig. 8: Relation between static values and types, and defunctionalisation environments and typing contexts, respectively

By assuming this relation between some defunctionalisation 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 normalisation of evaluation.

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 for the ability of defunctionalisation to determine uniquely 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 values maps to the same type. This property is used to establish that the types of order zero terms are unchanged by defunctionalisation. It is also used in the cases for conditionals, array literals, loops, and maps in the proof of Theorem 1.

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

Proof. By induction on the structure of sv. □

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

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

Proof. By induction on the typing derivation $\mathcal{T}$. Most cases are straightforward applications of the induction hypothesis to the subderivations, often reasoning by inversion on the obtained relations between static values and types, and extending the assumed relation $\vDash E : \Gamma$ to allow for further applications of the induction hypothesis. Then the required derivations are subsequently constructed directly. For details, please consult [23]. □

4.3 Preservation of Meaning

In this section, we show that the defunctionalisation 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 context Σ' as well (the notion of correspondence will be made precise shortly).

We first define a simple relation between source language values and static values, given in Figure 9, and extend it to relate evaluation environments and defunctionalisation environments in the usual way. Note that this relation actually defines a function from values to static values.

$\boxed{\vDash v : sv}$

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

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

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

Fig. 9: Relation between values and static values

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:

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

We extend this mapping homomorphically to evaluation environments. The case for arrays is actually moot, since arrays will never contain function closures.

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 used in the proof of type soundness, then the value maps to itself, that is, values that do not contain function closures are unaffected by defunctionalisation:

Lemma 4. *If* $\vDash v : \tau$ *and* τ orderZero*, then* $[\![v]\!]_{\text{val}} = v$.

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

We now prove the following theorem, which states that the defunctionalisation transformation preserves the meaning of an expression that is known to evaluate to some result, where the value of the defunctionalised 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* $r = \mathbf{err}$*, then also* $[\![\Sigma]\!]_{\text{val}} \vdash e' \downarrow \mathbf{err}$ *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 $\mathcal{E}$. For details, please consult [23]. □

4.4 Correctness of Defunctionalisation

To summarize the previous properties and results relating to the correctness of the defunctionalisation transformation, we state the following corollary which follows by type soundness (Lemma 1), normalisation and preservation of typing for defunctionalisation (Theorem 1), and semantics preservation of defunctionalisation (Theorem 2), together with Lemma 3 and Lemma 4.

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.*

5 Implementation

The defunctionalisation transformation that was presented in Section 3 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.

In this section, we discuss how our implementation diverges from the theoretical description. 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`, which are specially recognized by the compiler,

and exploited to perform optimisations 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 defunctionaliser is not *exclusively* first-order, but may contain fully saturated applications of these built-in functions.

5.1 Polymorphism, Function Types, and Monomorphisation

Futhark supports parametric let-polymorphism. Defunctionalisation, however, works only on monomorphic programs and therefore, programs are *monomorphized* before being passed to the defunctionaliser.

Due to our restrictions on function types, it is necessary to distinguish between type variables which may be instantiated with any type, 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 defunctionalise, for example by instantiating the type 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 situation from happening, we have introduced the notion of *lifted type variables*, written `'^a`, which are unrestricted in the types that they may be instantiated to, 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 Futhark equality and inequality operators `==` and `!=` are overloaded operators, which also work on structural types, such as arrays and tuples. However, Futhark does not support type classes [29] or equality types [11]. Allowing the equality and inequality operators to work on values of abstract types (i.e., on all non-lifted types) could potentially violate abstraction properties, which is the reason for the special treatment of equality types and equality type variables in the Standard ML programming language.

5.2 Array Shape Parameters

Futhark employs a system of runtime-checked size-dependent types, where the programmer may give shape declarations in function definitions to express shape invariants about parameter and result arrays. Shape parameters (listed before ordinary parameters and enclosed in brackets) are not explicitly passed on application. Instead, they are implicitly inferred from the arguments of the value parameters. Defunctionalisation could potentially destroy the shape invariants. For example, consider partially applying a function such as the following:

```
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. In the case of the function `f`, the defunctionalised 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 is useful to be able to specify shape invariants on expressions of function type in general. This feature can be implemented by eta-expanding the function expression and inserting type ascriptions with shape annotations on the order-zero parameters and bodies. For instance, the type ascription

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

would be translated into the expression

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

This feature has not yet been implemented in Futhark.

5.3 Optimisations

When the defunctionalisation algorithm processes an application, the D-App rule will replicate the lambda body (e_0) at the point of application. This implicit copying is equivalent to fully inlining all functions, which will produce very large programs if the same function is called in many locations. In our implementation, we instead perform lambda lifting [25] to move the definition of the lambda to a top-level function, parameterized by an argument representing its lexical closure, and simply insert a call to that function.

However, this lifting produces the opposite problem: we may now produce a very large number of trivial functions. In particular, when lifting curried functions that accept many parameters, we will create one function for each partial application, corresponding to each parameter. To limit the copying and lifting, our implementation extends the notion of static values with a *dynamic function*, which is simply a first-order functional analogue to dynamic values. We then add a translation rule similar to D-App that handles the case where the function is a dynamic function rather than a *Lam*.

Finally, our implementation inlines lambdas with particularly simple bodies; in particular those that contain just a single primitive operation or a record literal. The latter case corresponds to functions produced for partial applications.

6 Empirical Evaluation

The defunctionalisation technique presented in this paper can be empirically evaluated by two metrics. First, is the code produced by defunctionalisation efficient? Second, are higher-order functions with our type restrictions useful? The

former question is the easier to answer, as we can simply rewrite a set of benchmark programs to make use of higher-order functions, and measure whether the performance of the generated code changes. We have done this by using the existing Futhark benchmark suite, which contains more than thirty Futhark programs translated from a range of other suites, including Accelerate [4], Rodinia [7], Parboil [32], and FinPar [1]. These implementations all make heavy use of operations such as `map`, `reduce`, `scan`, `filter`, which used to be language constructs, but are now higher-order functions that wrap compiler intrinsics. Further, most benchmarks have been rewritten to make use of higher-order utility functions (such as `flip`, `curry`, `uncurry`, and function composition and application) where appropriate. As expected, this change had no impact on run-time performance, although compilation times did increase by up to a factor of two.

The more interesting question is whether the restrictions we put on higher-order functions are too onerous in practice. While some uses of higher-order functions are impossible, many "functional design patterns" are unaffected by the restrictions. Such examples include the use of higher-order functions for defining a Futhark serialisation library [27,13] and for introducing the notion of functional images [9], as we shall see in the following section. Higher-order functions also make it possible to capture certain reusable parallel design patterns, for instance, for flattening some cases of nested irregular parallelism [15].

6.1 Functional Images

Church Encoding can be used to represent objects such as integers via lambda terms. While modern functional programmers tend to prefer built-in numeric types for efficiency reasons, other representations of data as functions have remained popular. One of these is functional images, as implemented in the Haskell library Pan [9]. Here, an image is represented as a function from a point on the plane to some value. In higher-order Futhark, we can define this as

```
type img 'a = point -> a
type cimage = img color
```

for appropriate definitions of `point` and `color`. Transformations on images are then defined simply as function composition.

Interestingly, none of the combinators and transformations defined in Pan require the aggregation of images in lists, or returning them from a branch. Hence, we were able to translate the entirety of the Pan library to Futhark. The reason is likely that Pan itself was designed for staged compilation, where Haskell is merely used as a meta-language for generating code for some high-performance object language [10]. This approach requires restrictions on the use of functions that are essentially identical to the ones we introduced for Futhark. In Futhark, we can directly generate high-performance parallel code, and modern GPUs are easily powerful enough to render most functional images (and animations) at a high frame rate. Essentially, once the compiler finishes its optimisations, we are left with a trivial two-dimensional `map` that computes the color of each pixel

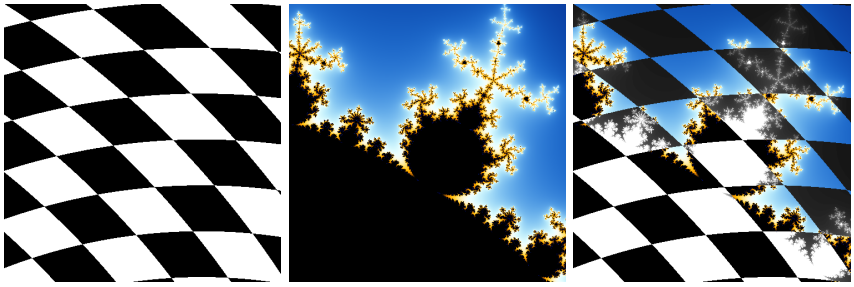

Fig. 10: Images rendered by the Futhark implementation of functional images. The annulus defined by the left-most image is used to overlay grey scale and colorized Mandelbrot fractals.

completely independently. Example images are shown on Figure 10. The Mandelbrot fractal, the implementation of which is translated from [26], in particular is expensive to compute at high resolutions.

7 Allowing Conditionals of Function Type

Given that the main novelty enabling efficient defunctionalisation 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 restriction on the type of conditionals. This change introduces a binary choice for the static value of a conditional and this choice may depend on dynamic information. The produced static value must capture this choice. Thus, we may extend the definition of static values as follows:

$$sv ::= \cdots \mid \mathit{Or}\ sv_1\ sv_2$$

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. 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-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.

8 Related Work

Support for higher-order functions is not widespread in parallel programming languages. For example, they are not supported in the pioneering work on NESL [3], which was targeted at a vector execution model with limitations similar to modern GPUs. Data Parallel Haskell (DPH) [5] does support higher-order functions via closure conversion, but targets traditional multicore CPUs where this is a viable technique. The GPU language Harlan [22] is notable for its powerful feature set, and it does support higher-order functions via Reynolds-style defunctionalisation. The authors of Harlan note that this could cause performance problems, but that it has not done so yet. This is likely because most of the Harlan benchmark programs do not make much use of closures on the GPU.

A general body of related work includes mechanisms for removing abstractions at compile time including the techniques, used for instance by Accelerate [4] and Obsidian [8], for embedded domain specific languages (EDSLs). These languages use a staged compilation approach where Haskell is used as a meta-language to generate first-order imperative target programs. While the target programs are themselves first-order, meta-programs may use the full power of Haskell, including higher-order functions. As our approach has limitations, so does the EDSL approach; in particular, care has to be taken that source language functions do not end up in target arrays. Other approaches at removing abstractions at compile time include the use of quoted domain specific languages [28], techniques for multi-stage programming, such as [33], and the notion of static interpretation of modules [12], which is also applied in the context of Futhark [2,14] for eliminating even higher-order module language constructs entirely at compile time (before monomorphisation).

Another body of related work includes the seminal work by Tait [34] and Girard [16] on establishing the basic proof technique on using logical relations for expressing normalisation and termination properties for the simply-typed lambda calculus and System F, which has been the inspiring work for establishing the property of termination for our defunctionalisation technique.

9 Conclusion and Future Work

We have shown a useful design for implementing higher-order functions in high-performance functional languages, by using a defunctionalisation transformation that exploits type-based restrictions on functions to avoid introducing branches in the resulting first-order program. We have proven this transformation correct. Further, we have discussed the extensions and optimisations we found necessary for applying the transformation in a real compiler, and demonstrated that the type restrictions are not a great hindrance in practice.

References

1. Andreetta, C., Bégot, V., Berthold, J., Elsman, M., Henglein, F., Henriksen, T., Nordfang, M.B., Oancea, C.E.: Finpar: A parallel financial benchmark. ACM Transactions on Architecture and Code Optimization (TACO) **13**(2), 18:1–18:27 (June 2016)
2. Annenkov, D.: Adventures in Formalisation: Financial Contracts, Modules, and Two-Level Type Theory. Ph.D. thesis, University of Copenhagen (April 2018)
3. Blelloch, G.E.: Programming Parallel Algorithms. Communications of the ACM (CACM) **39**(3), 85–97 (1996)
4. Chakravarty, M.M., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating Haskell array codes with multicore GPUs. In: Workshop on Declarative Aspects of Multicore Programming. DAMP '11, ACM (January 2011)
5. Chakravarty, M.M., Leshchinskiy, R., Jones, S.P., Keller, G., Marlow, S.: Data Parallel Haskell: A Status Report. In: Workshop on Declarative Aspects of Multicore Programming. DAMP '07, ACM (January 2007)
6. Charguéraud, A.: Pretty-big-step semantics. In: Proceedings of the 22nd European Conference on Programming Languages and Systems. pp. 41–60. ESOP '13, Springer-Verlag, Berlin, Heidelberg (March 2013)
7. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization. IISWC '2009 (October 2009)
8. Claessen, K., Sheeran, M., Svensson, B.J.: Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In: Workshop on Declarative Aspects of Multicore Programming. DAMP '12, ACM (January 2012)
9. Elliott, C.: Functional images. In: The Fun of Programming. "Cornerstones of Computing" series, Palgrave (Mar 2003)
10. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. Journal of Functional Programming **13**(2) (2003)
11. Elsman, M.: Polymorphic equality—no tags required. In: Second International Workshop on Types in Compilation (TIC'98) (March 1998)
12. Elsman, M.: Static interpretation of modules. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming. ICFP '99, ACM Press (September 1999)
13. Elsman, M.: Type-specialized serialization with sharing. In: Sixth Symposium on Trends in Functional Programming (TFP'05) (September 2005)
14. Elsman, M., Henriksen, T., Annenkov, D., Oancea, C.E.: Static interpretation of higher-order modules in Futhark: Functional GPU programming in the large. Proceedings of the ACM on Programming Languages **2**(ICFP), 97:1–97:30 (Jul 2018)

15. Elsman, M., Henriksen, T., Oancea, C.E.: Parallel Programming in Futhark. Department of Computer Science, University of Copenhagen (November 2018), https://futhark-book.readthedocs.io
16. Girard, J.Y.: Interpretation Fonctionnelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur. In: Proceedings of the Second Scandinavian Logic Symposium. pp. 63–92. North-Holland (1971)
17. Henriksen, T.: Design and Implementation of the Futhark Programming Language. Ph.D. thesis, DIKU, University of Copenhagen (November 2017)
18. Henriksen, T., Elsman, M., Oancea, C.E.: Size slicing: a hybrid approach to size inference in Futhark. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional High-Performance Computing. FHPC '14, ACM (2014)
19. Henriksen, T., Elsman, M., Oancea, C.E.: Modular acceleration: Tricky cases of functional high-performance computing. In: Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing. FHPC '18, ACM, New York, NY, USA (September 2018)
20. Henriksen, T., Serup, N.G., Elsman, M., Henglein, F., Oancea, C.E.: 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. pp. 556–571. PLDI '17, ACM (June 2017)
21. Henriksen, T., Thore, F., Elsman, M., Oancea, C.E.: Incremental flattening for nested data parallelism. In: Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '19, ACM (2019)
22. Holk, E., Newton, R., Siek, J., Lumsdaine, A.: Region-based memory management for GPU programming languages: Enabling rich data structures on a spartan host. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. pp. 141–155. OOPSLA '14, ACM, New York, NY, USA (October 2014)
23. Hovgaard, A.K.: Higher-order functions for a high-performance programming language for GPUs. Master's thesis, Department of Computer Science, Faculty of Science, University of Copenhagen, Universitetsparken 5, DK-2100 Copenhagen (May 2018)
24. Hughes, J.: Why Functional Programming Matters. The Computer Journal **32**(2), 98–107 (1989)
25. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Proc. of Conference on Functional Programming Languages and Computer Architecture. Springer-Verlag, New York, NY, USA (1985)
26. Jones, M.P.: Composing fractals. Journal of Functional Programming **14**(6), 715–725 (November 2004)
27. Kennedy, A.J.: Functional pearl: Pickler combinators. Jounal of Functional Programming **14**(6), 727–739 (Nov 2004)
28. Najd, S., Lindley, S., Svenningsson, J., Wadler, P.: Everything old is new again: Quoted domain-specific languages. In: Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation. PEPM '16, ACM (January 2016)
29. Peterson, J., Jones, M.: Implementing type classes. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. pp. 227–236. PLDI '93, ACM, New York, NY, USA (1993)
30. Poulsen, C.B., Mosses, P.D.: Flag-based big-step semantics. Journal of Logical and Algebraic Methods in Programming **88**, 174 – 190 (2017)
31. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the ACM annual conference-Volume 2. pp. 717–740. ACM (1972)

32. Stratton, J.A., Rodrigues, C., Sung, I.J., Obeid, N., Chang, L.W., Anssari, N., Liu, G.D., Hwu, W.m.W.: Parboil: A revised benchmark suite for scientific and commercial throughput computing. Tech. rep., University of Illinois at Urbana-Champaign (2012), IMPACT-12-01
33. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theoretical Computer Science **248**(1), 211–242 (2000), PEPM '97
34. Tait, W.W.: Intensional interpretations of functionals of finite type. Journal of symbolic logic **32**, 198–212 (1967)