Under consideration for publication in J. Functional Programming

Semantics of Functional Data-Parallelism: from an applied λ-calculus to a distributed shared memory implementation

Artjoms Šinkarovs and Sven-Bodo Scholz
Heriot-Watt University, Scotland, UK
(*e-mail:* `a.sinkarovs@hw.ac.uk, s.scholz@hw.ac.uk`)

Abstract

We demonstrate three successive refinements of the operational semantics for an array-based data-parallel language. We start with an abstract semantics, add the concept of storages and reference counting and, finally, derive semantics for shared-memory and distributed-memory executions. We also formalise the criteria for destructive updates and demonstrate that they hold under both versions of parallel semantics. Successive refinements make it possible to formally verify that the initial semantics is preserved throughout. The obtained semantics can serve as an implementation guide of the runtime system for data-parallel languages. The distributed semantics that we obtain at the end maps nicely onto Distributed Shared Memory (DSM). It turns out that the functional setting here facilitates a DSM implementation with special properties that enables an implementation with particularly low runtime overhead.

Contents

1 Introduction 2
2 Applied lambda calculus for data parallelism 4
2.1 Relation to SAC 6
2.2 Type system 7
2.3 Substitution Semantics 8
2.4 Abstract semantics 9
2.5 Towards a compiler for λ_{DP} 10
3 Semantics with storages 10
3.1 Map/reduce with non-deterministic order 14
3.2 Reuse criteria 18
3.3 Trade-offs 20
4 Shared memory semantics 21
4.1 Concurrency in the language 21
4.2 Distributions 21
4.3 Rules for shared-memory semantics 22
5 Semantics for distributed memory 26
5.1 Reuse under distributed semantics 31
5.2 Implementation considerations 31

6 Related Work 32
7 Conclusions and future work 33
References 34

1 Introduction

Data parallelism has always featured prominently in the context of High-Performance Computing (HPC). Auto-parallelising compilers as well as most programming languages that target high degrees of parallelism are based on data-parallel constructs. This includes long-established languages, such as Fortran90 and HPF, as well as more recent developments like CoArray Fortran and Chapel.

Over the last decade, the functional programming community followed this trend by taking significant interest in data parallelism — partly due to the need to utilise massively parallel hardware effectively for most of which data parallelism gives a very suitable level of abstraction. This led to the creation of several languages, DSLs, and language extensions for expressing data parallelism in a declarative way. Examples include: SISAL (Feo *et al.*, 1990), NESL (Blelloch, 1992), SAC (Grelck & Scholz, 2006), DPH (Chakravarty *et al.*, 2007), Accelerate (Chakravarty *et al.*, 2011), Obsidian (Svensson *et al.*, 2010), Feldspar (Axelsson *et al.*, 2011), Diderot (Chiw *et al.*, 2012), and Futhark (Henriksen *et al.*, 2014). They all build on the conceptual advantages of a side-effect-free setting and advanced type systems, allowing compilers for these languages to apply more radical program transformations than their imperative counterparts.

Several of these languages provide multiple back-ends that enable them to generate codes for different execution platforms, typically including shared-memory multi-cores and GPUs as obvious targets, and FPGAs or even distributed-memory clusters in more advanced cases. When building compilers for such languages, it often turns out to be crucial to leverage very subtle techniques to obtain reasonably good performance. As the hardware landscape becomes increasingly diverse with quickly changing core features such as bandwidth and latency of data buses, memories or communication facilities, the compilation into efficient code becomes even more challenging. In turn, more intricate program transformations become necessary adding to the challenge of ensuring the correctness of these transformations.

By correctness we mean formal guarantees that compiled programs always evaluate to the same values as the semantics of their original counterparts dictates. To construct a proof for this one has to relate evaluations of compiled programs to those of the corresponding original programs and to demonstrate that they match. Usually, this is very difficult to do in a single step, as levels of abstraction of the source language and the target language are vastly different. Added difficulty arises from the fact that many languages, in particular languages closer to the hardware level such as C, allow for programs that have undefined behaviours. Therefore, as we can observe it at the example of CompCert (Leroy, 2009), such proofs are performed via multiple transformation steps, successively lowering the level of abstraction, making it closer to the underlying hardware of interest. After that, individual transformations are verified.

ZU064-05-FPR paper 17 March 2020 16:57

When a transformation happens within the same language, the proof of correctness is relatively straight-forward. However, very often this is not the case. Not only the transformation output language may change but also, semantics of the output language may introduce some novel concepts that were not present in the original semantics, *e.g.* threads, mutexes, sending data over network, *etc*. In those cases, reasoning about correctness gets severely more complicated.

The usual approach to this problem is proof by bisimulation based on trace semantics of the input and output languages. This means that traces have to be introduced and analysed in the proofs, which very quickly gets unbearably complicated. Also, this approach is quite resistant to changes in the overall language design. Which makes this approach not very suitable for actively developing languages.

To make a proof more generic, we observe that if semantics of the target language is specific enough, then deriving correct implementation is rather straight forward. Our key idea in this paper is to shift the understanding of a compilation process from classical "transforming a program in language A into a program in a language B" into "transition from evaluation under semantics α to evaluation under semantics β". In this new paradigm the essence of the proof that compilation is correct changes as well. We now have to demonstrate equivalences of two semantics rather than equivalence of two programs in different languages. Such an approach raises a number of questions:

1. would it be possible to encode all the differences between intermediate representations A and B into novel semantics?
2. can we do this for all types of hardware of interest?
3. how to demonstrate equivalences of two semantics?

To find some of the answers in the context of functional data-parallel languages, we build a formalism that is based on λ-calculus enriched with explicit data-parallel constructs. The main feature of this formalism is its conciseness. We reduce the number of constructs to absolute minimum, yet keeping it easily extendable and making sure that it reflects essential difficulties that come when compiling data parallel languages.

We consider practical application of the proposed approach in the context of the SAC programming language and its compiler `sac2c`. Currently `sac2c` has backends for multicore CPUs and GPUs; the backend for distributed systems is work in progress. Distributed backend for a data parallel language can be designed in a number of different ways all of which come with their specific trade-offs. Our current hypothesis is that functional array-based data parallel languages are very much suitable for using distributed shared memory to tackle cluster architectures. By suitable we mean that distributed shared memory can be used with very little overheads.

To substantiate this claim and to demonstrate mechanisms that will be needed, we refine semantics of our minimalistic formalism so that it could operate on distributed systems. From there we can study essential mechanisms that will be needed to implementations this idea in the context of SAC.

It turns out that reasoning about correctness of the distributed semantics gets much simpler if we introduce intermediate forms of the semantics of our minimalistic language, capture aspects such as memory, multiple threads and, eventually, multiple nodes of a distributed-memory system. As a result we get a formal proof that, in the functional setting, a feasible distributed-shared-memory implementation is possible, and that it provides consistency upon synchronisation, without requiring any exchange of written data.

As it turns out, all the intermediate forms of the semantics we develop correspond well with the subtleties of the corresponding underlying implementations in the context of SAC. Despite this strong resemblance to real implementations, the formalisms needed are relatively concise. In fact, they can be seen as a blueprint for very efficient implementations on single-core, shared-memory, multi-core and distributed-memory clusters for any data parallel language that can be stripped down to the proposed formalism.

In detail, the contributions of our paper are:

1. a big-step semantics for an applied λ-calculus enriched with data parallel constructs; an extension of the semantics that captures the notion of storage and dynamic support for reference counting; a variant of the semantics that captures multi-threaded; execution on shared-memory systems (the distinguishing feature of this variant is that it nicely captures the reference counting issues involved in parallel execution); a big-step semantics that captures SPMD-style execution on distributed memory systems;
2. relation to the existing implementation at the example of `sac2c`; and
3. proof sketches for the equivalence of all these semantics.

The paper is organised as follows: Section 2 introduces our prototypical data-parallel λ-calculus λ_{DP}, defines the language and its abstract semantics, and presents some initial implementation aspects of the language. Section 3 adds the notion of storage and reference counting to the semantics for λ_{DP}. Section 4 introduces the notion of parallel execution, and discusses scheduling issues as well as the implications for parallel reference counting. The final extension of the semantics is presented in Section 5. Here, the notion of distributed memory is captured and an extensive discussion about the implications for a distributed, shared memory implementation is provided. Finally, we discuss related work in Section 6 before Section 7 concludes.

2 An applied λ-calculus for data parallelism

In this section, we present an applied λ-calculus called λ_{DP}. It is designed to be as minimal as possible, while capturing the essence of data parallelism. λ_{DP} supports arrays and data-parallel operations by means of *imap* and *reduce* operators as first-class citizens. As primitive operators, we introduce arithmetic and comparison operations on scalars, and random-access selections into arrays. Conditionals and *letrec* give rise to recursive functions. We restrict ourselves to one-dimensional arrays. An extension to multi-dimensional arrays is possible via nesting (). The dimensionality of any nested array must be known at compiler time, therefore rank-polymorphic operations are not supported. This restriction can be relaxed, as it is demonstrated at the example of SAC, but the price for this would be far less elegant formalism. From the semantics perspective, there will be no further insights into the model as at runtime, dimensionalities of all the arrays will be known.

We define the syntax of λ_{DP} as follows:

$$\begin{array}{rcll}
e & ::= & c & \text{(constants)} \\
 & \mid & x & \text{(variables)} \\
 & \mid & \lambda x.e & \text{(abstractions)} \\
 & \mid & e\ e & \text{(applications/selections)} \\
 & \mid & \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e & \text{(conditionals)} \\
 & \mid & \mathit{letrec}\ x = e\ \mathit{in}\ e & \text{(conditionals)} \\
 & \mid & \mathit{imap}\ e\ e & \text{(index map)} \\
 & \mid & \mathit{reduce}\ e\ e\ e & \text{(reduce)} \\
 & \mid & e + e, \ldots & \text{(arithmetic/logical operations)} \\
 & & & \\
c & ::= & 0, 1, \pi, 42, \ldots & \text{(numbers)} \\
 & \mid & \mathit{true}, \mathit{false} & \text{(boolean values)} \\
 & \mid & \{c \mapsto c, \ldots, c \mapsto c\} & \text{(arrays)}
\end{array}$$

Constants are either scalar numbers or arrays. Arrays are mappings of scalar indices into values, where all the indices of an array are distinct integer numbers. As an example, consider:

$$a = \{0 \mapsto 7, 5 \mapsto 3, 1 \mapsto 6\}$$

It describes an array a which is defined for indices 0, 1, and 5. Corresponding selections denoted as applications to indices a 0, a 1, and a 5, result in the values 7, 6, and 3, respectively. This effectively provides λ_{DP} with associative arrays which we have chosen mainly for notational convenience when it comes to expressing partitioning. In most of the existing languages, *e.g.* SAC, arrays are dense with respect to their index ranges, which makes it possible to guarantee $O(1)$ selection. Dense arrays are straight-forwardly convertible to associative arrays, therefore our formalism is dealing with richer data structures, yet capturing the dense case which shall be implied when we refer to implementations.

Our formalism uses two meta-operators: for any given array a, $|a|$ provides the number of elements a has and $I(a)$ denotes the set of indices of a. For the array from the previous example, we have:

$$|a| = 3 \qquad\qquad I(a) = \{0, 5, 1\}$$

Lambda abstractions and function applications are defined in the usual way; brackets may be inserted for disambiguation.

The most important extensions to our applied λ-calculus are array-based variants of *map*/*reduce* combinators. The *reduce* combinator is defined as usual: it folds all arguments of an array into a single value. Instead of classical *map* we are going to use the *i*ndex *map* called *imap*. Both combinators apply a function provided as first argument to all array elements given as the second argument. The difference is that *imap* applies the function to every index of an array, whereas *map* applies it to every value. Consider the following example:

$$\mathit{map}\ (\lambda x.x + 1)\ \{0 \mapsto 7, 5 \mapsto 3\} \Downarrow \{0 \mapsto 8, 5 \mapsto 4\}$$

$$\mathit{imap}\ (\lambda i.i + 1)\ \{0 \mapsto 7, 5 \mapsto 3\} \Downarrow \{0 \mapsto 1, 5 \mapsto 6\}$$

It turns out that when random-access selections are supported, the box-standard *map* operator is not potent enough to conveniently and concisely describe any operations on arrays where the result values depend on their index positions. As an example, consider a function reverse which inverses the order of the elements in an array. With *imap*, we can define this function as:

$$\text{reverse} \equiv \lambda a.imap\ \lambda i.(a\ ((\text{len}\ a) - i - 1))\ a$$

assuming that the indices of a are enumerated from 0 to $|a| - 1$ without gaps, and assuming that the function $(\text{len}\ a)$ computes $|a|$. In order to achieve the same result using a single *map*, one would have to construct an array where indices of the array we want to reverse appear as *values*. This makes a definition of reverse in terms of *map* clumsy and inefficient: we would have to create extra array and use an indirect access for every element.

Note that when random access selections are built-in, our intuition of *map*-like constructs shall change. Conceptually, *map* and *imap* are not recursively defined higher-order functions applying *head* and *tail*, but rather an operator that combines results of n function applications into an array. All the applications are truly independent, *i.e.* no order is attached.

Finally, *imap* in its presented form makes *map* redundant as we have for arbitrary expressions f and a:

$$map\ f\ a\ =\ imap\ \lambda i.(f\ (a\ i))\ a$$

assuming that i does not occur in the free variables of f or a. These observations allow us to avoid *map* in our language.

The *reduce* construct takes a two-argument function as its first argument, which is applied successively to the values of the third argument, using second argument as neutral element. Consider the following reduce statement, assuming that f sums its arguments — $f \equiv \lambda x.\lambda y.x + y$:

$$reduce\ f\ 0\ \{0 \mapsto 7, 5 \mapsto 3, 1 \mapsto 6\} \equiv f\ 7\ (f\ 3\ (f\ 6\ 0)) \Downarrow 16$$

We use the 0, the neutral element of f, to make sure that *reduce* evaluates to some value even if the second argument of reduce contains a single element or is an empty array.

2.1 Relation to SAC

As stated in the introduction already, the design of λ_{DP} aims at constituting a generic core suitable for describing various aspects for implementing functional array languages in general. Throughout the paper we try to substantiate that claim by relating it to one concrete such language, namely to SAC.

As mentioned earlier, SAC only supports dense arrays starting with index 0. This restriction can easily be statically checked since the only way to construct an index range in λ_{DP} is by means of a constant array. Another difference between λ_{DP} and SAC is the support for n-dimensional arrays and array operations in SAC. While mapping n-dimensional arrays into one-dimensional ones requires some advanced analyses and program transformations, *e.g.* (**?**; **?**) , it does not add any further insights for the semantics-transformations discussed in the remainder of the paper.

When looking at the core language constructs, we can observe that the *imap* operator of λ_{DP} indeed rather closely resembles the *with*-loop in SAC. The main difference here being that in SAC there is a built-in facility for specifying index ranges which, in SAC, are referred-to as generators. SAC's distinction between *genarray* and *modarray with*-loops constitutes a syntactic sugar which in the context of λ_{DP} vanishes completely. The *fold with*-loops of SAC are equivalent to the *reduce* operation of λ_{DP}.

2.2 Type system

We introduce a type system for our language based on the types from $\mathbb{T}$ defined inductively:

$$\textsc{Builtin-type}\ \frac{t \in \{\textbf{int}, \textbf{float}, \textbf{bool}\}}{\mathbb{T} \vdash t} \qquad \textsc{Fun-type}\ \frac{\mathbb{T} \vdash A \qquad \mathbb{T} \vdash B}{\mathbb{T} \vdash A \to B} \qquad \textsc{Array-type}\ \frac{\mathbb{T} \vdash A \in \{\textbf{int}, \textbf{float}, \textbf{bool}\}}{\mathbb{T} \vdash [A]}$$

We restrict the type of array arguments to scalars because it simplifies our formalism; it is not a conceptual restriction. See Section 3.3 for further discussion on λ_{DP} extensions. The typing context is defined as usual:

$$\Gamma ::= \cdot \mid \Gamma, x \mapsto A \qquad A \in \mathbb{T}$$

We use the following typing rules, assuming that $\mathscr{K}$ is a mapping from constants and built-in functions to types:

$$\textsc{Const}\ \frac{}{\Gamma \vdash c : \mathscr{K}(c)} \qquad \textsc{Prf}\ \frac{\oplus \in \{+, -, \dots\}}{\Gamma \vdash \oplus : \mathscr{K}(\oplus)} \qquad \textsc{Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Sel}\ \frac{\Gamma \vdash a : [A] \qquad \Gamma \vdash i : \textbf{int}}{\Gamma \vdash a\ i : A}$$

$$\textsc{Array}\ \frac{\begin{array}{ccc}\Gamma \vdash i_1 : \textbf{int} & \dots & \Gamma \vdash i_n : \textbf{int} \\ \Gamma \vdash e_1 : A & \dots & \Gamma \vdash e_n : A\end{array}}{\Gamma \vdash \{i_1 \mapsto e_1, \dots, i_n \mapsto e_n\} : [A]} \qquad \textsc{Abs}\ \frac{\Gamma, x \mapsto A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B} \qquad \textsc{App}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : A \to B \\ \Gamma \vdash e_2 : A\end{array}}{\Gamma \vdash e_1\ e_2 : B}$$

$$\textsc{IMap}\ \frac{\Gamma \vdash f : \textbf{int} \to B \qquad \Gamma \vdash a : [A]}{\Gamma \vdash \textit{imap}\ f\ a : [B]} \qquad \textsc{Reduce}\ \frac{\Gamma \vdash f : A \to A \to A \qquad \Gamma \vdash e_{\text{neut}} : A \qquad \Gamma \vdash a : [A]}{\Gamma \vdash \textit{reduce}\ f\ e_{\text{neut}}\ a : A}$$

$$\textsc{Cond}\ \frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : A \qquad \Gamma \vdash e_3 : A}{\Gamma \vdash \textit{if}\ e_1\ \textit{then}\ e_2\ \textit{else}\ e_3 : A} \qquad \textsc{LetRec}\ \frac{\Gamma \vdash e_1 : A \qquad \Gamma, x \mapsto A \vdash e_2 : B}{\Gamma \vdash \textit{letrec}\ x = e_1\ \textit{in}\ e_2 : B}$$

The type rules are commonplace, and besides the usual guarantees that functions and arguments match at function applications, the predicate of a conditional is boolean and branches have the same type, the type of *letrec*'s bound expression is consistent with the body, the type system also guarantees that types of all the array elements are the same; array selections are typeable similarly to function applications; *imap*'s first argument is a function of the correct type, and *reduce*'s first argument is a two-argument function that matches the neutral element and the element type of the argument array. The type signature of *reduce*'s function does not fix left or right order, allowing to choose any implementation under assumption that the function is associative.

2.3 Substitution Semantics

One of the most compact operational semantics for λ_{DP} is a big-step operational semantics based on the concept of substitution. Here is how it can look like:

$$\textsc{Const}\ \frac{}{c \Downarrow c} \qquad \textsc{Prf}\ \frac{\oplus \in \{+,-,\dots\}}{\oplus \Downarrow \oplus} \qquad \textsc{Abs}\ \frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \textsc{App}\ \frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v \qquad e[v/x] \Downarrow v_1}{e_1\ e_2 \Downarrow v_1}$$

$$\textsc{Cond-True}\ \frac{e_1 \Downarrow \mathit{true} \qquad e_2 \Downarrow v}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow v} \qquad \textsc{Cond-False}\ \frac{e_1 \Downarrow \mathit{false} \qquad e_3 \Downarrow v}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow v}$$

$$\textsc{LetRec}\ \frac{e_1 \Downarrow v \qquad v_1 = v[(\mathit{letrec}\ x = e_1\ \mathit{in}\ x)/x] \qquad e_2[v_1/x] \Downarrow v_2}{\mathit{letrec}\ x = e_1\ \mathit{in}\ e_2 \Downarrow v_2} \qquad \textsc{Sel}\ \frac{a \Downarrow \{i_1 \mapsto v_1, \dots, i_n \mapsto v_n\} \qquad i \Downarrow v \qquad \exists k \in \{1,\dots,n\} : i_k = v}{a\ i \Downarrow v_k}$$

$$\textsc{IMap}\ \frac{a \Downarrow \{i_1 \mapsto v_1, \dots, i_n \mapsto v_n\} \qquad \forall k \in \{1,\dots,n\} : f\ i_k \Downarrow v'_k}{\mathit{imap}\ f\ a \Downarrow \{i_1 \mapsto v'_1, \dots, i_n \mapsto v'_n\}} \qquad \textsc{Reduce}\ \frac{a \Downarrow \{i_1 \mapsto v_1, \dots, i_n \mapsto v_n\} \qquad f\ v_1\ (f\ \dots (f\ v_n\ e_{\text{neut}}) \dots) \Downarrow v}{\mathit{reduce}\ f\ e_{\text{neut}}\ a \Downarrow v}$$

We use $e[v/x]$ to denote capture-avoiding substitution of free occurrences of x in the expression e with v.

The values in this semantics are constants and abstractions (which include primitive functions). The rules for abstraction, application and conditions are commonplace. Letrec uses double substitution to achieve a fixpoint. For the $\mathit{letrec}\ f = \lambda x.E[f]\ \mathit{in}\ e$ expression, v and v_1 will be:

$$v = \lambda x.E[f] \qquad\qquad v_1 = \lambda x.E[\mathit{letrec}\ f = \lambda x.E[f]\ \mathit{in}\ f]$$

Where $E[f]$ denotes an expression where f occurs free. Selections return the value bound at index i in the array a. Imap applies f for every index of the array a combining results in the new array. Reduce, for conciseness, is expressed as right fold, although under assumption that f is associative the order of applications of f can be different.

Such a semantics is very concise, but it is far away from a realistic implementation. The reason for this lies in using substitution: when evaluating applications and letrecs, we traverse the entire term to make a substitution and then we traverse it again to do the actual evaluation. Also, in case of recursive *letrec*, there is no need to reevaluate the substituted *letrec* expression on every recursive unfolding.

2.4 Abstract semantics

To deal with those problems we lower down the level of abstraction in our semantics and we introduce the notion of environments that will help to delay the substitution up to the moment we actually need the substituted expression. To define this semantics, we use a *natural semantics* similar to the one described in (Kahn, 1987). A judgement takes the form:

$$\rho \vdash e \Downarrow v$$

where ρ is an environment and v is the result of evaluation of the expression e in ρ. The environment ρ is an ordered list of variable-value bindings, separated by commas:

$$\rho ::= \emptyset \mid \rho, x \mapsto v$$

the environment lookup, denoted $\rho(x)$, is evaluated from *right to left*. The values in the current semantics are arrays and scalar constants, or function closures denoted as $[\lambda x.e, \rho]$:

$$v ::= c \mid [\lambda x.e, \rho]$$

The core rules of the abstract semantics of λ_{DP} are:

$$\textsc{Const}\ \frac{}{\rho \vdash c \Downarrow c} \qquad \textsc{Var}\ \frac{x \in \rho}{\rho \vdash x \Downarrow \rho(x)} \qquad \textsc{Abs}\ \frac{}{\rho \vdash \lambda x.e \Downarrow \llbracket \lambda x.e, \rho \rrbracket}$$

$$\textsc{Sel}\ \frac{\rho \vdash i \Downarrow v \qquad \rho \vdash a \Downarrow \{i_1 \mapsto v_1, \ldots, i_n \mapsto v_n\} \qquad \exists k \in \{1, \ldots, n\} : i_k = v}{\rho \vdash a\ i \Downarrow v_k}$$

$$\textsc{App}\ \frac{\rho \vdash e_1 \Downarrow \llbracket \lambda x.e, \rho_1 \rrbracket \qquad \rho \vdash e_2 \Downarrow v \qquad \rho_1, x \mapsto v \vdash e \Downarrow w}{\rho \vdash e_1\ e_2 \Downarrow w}$$

$$\textsc{Prf}\ \frac{\oplus \in \{+, -, \ldots\} \qquad \rho \vdash e_1 \Downarrow v_1 \qquad \rho \vdash e_2 \Downarrow v_2 \qquad v_1\, \text{sem}(\oplus)\, v_2 = w}{\rho \vdash e_1 \oplus e_2 \Downarrow w}$$

$$\textsc{Cond-True}\ \frac{\rho \vdash e_1 \Downarrow \mathit{true} \qquad \rho \vdash e_2 \Downarrow v}{\rho \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow v}$$

$$\textsc{Cond-False}\ \frac{\rho \vdash e_1 \Downarrow \mathit{false} \qquad \rho \vdash e_3 \Downarrow v}{\rho \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \Downarrow v} \qquad \textsc{LetRec}\ \frac{\rho, x \mapsto v \vdash e_1 \Downarrow v \qquad \rho, x \mapsto v \vdash e_2 \Downarrow v_1}{\rho \vdash \mathit{letrec}\ x = e_1\ \mathit{in}\ e_2 \Downarrow v_1}$$

Note that in the PRF rule, 'sem($\oplus$)' refers to the semantics of $\oplus$ defined by λ_{DP}. The rule for *letrec* now leaves an implementation of the potential fixpoint open. The only assertion here is that the value bound to x is identical to the result of the evaluation of e_1.

On top of the core rules we have the rules for the data-parallel operators *imap* and *reduce*:

$$\textsc{Reduce}\quad \frac{\rho \vdash a \Downarrow \{i_1 \mapsto v_1, \ldots, i_n \mapsto v_n\} \qquad \rho \vdash f\ v_1\ (f\ \ldots (f\ v_n\ e_{\text{neut}}) \ldots) \Downarrow v}{\rho \vdash \mathit{reduce}\ f\ a \Downarrow v}$$

$$\textsc{IMap}\quad \frac{\rho \vdash a \Downarrow \{i_1 \mapsto v_1, \ldots, i_n \mapsto v_n\} \qquad \forall k \in \{1, \ldots, n\} : \rho \vdash f\ i_k \Downarrow v'_k}{\rho \vdash \mathit{imap}\ f\ a \Downarrow \{i_1 \mapsto v'_1, \ldots, i_n \mapsto v'_n\}}$$

The REDUCE rule, as in the previous semantics, defines the order of applications of f to the elements of the evaluated array a. However, if f is associative and commutative, then reductions can be performed in arbitrary order. We exploit this fact in later refinements of the semantics.

2.5 Towards a compiler for λ_{DP}

Construction of a compiler from the semantics without substitutions presented here is straightforward. As we have one rule per syntactical construct, the deduction tree, if it exists, can be derived by simply following the syntactical construction of the program and by choosing an order in which to compute the pre-conditions of each rule that observes any given dependencies. We have to decide how exactly one implements the *letrec* construct, but apart from that, one can use a standard applicative order evaluator where the environment can be implemented through a cactus stack (Hauck & Dent, 1968) to cater for closures.

There is a lot of optimization potential in *imap*: The array argument values are not needed; only the indices are of interest. This often allows the second argument to be optimized away. If partial evaluation provides knowledge about the array's index space, the second argument to *imap* can be replaced by a generator.

We also observe that mapping the environment into a cactus stack is efficient only when we look at scalar data. As soon as arrays are put into the environment, creating copies when applying the VAR rule is prohibitively expensive, particularly when such arrays may be repeatedly pushed back into the environment through the APP rule. Any practical implementation needs to avoid this overhead, which immediately leads to our next refinement of the semantics.

3 Semantics with storages

The key concept in avoiding array is to introduce the concept of sharing, *i.e*. pointers. We change our environments to hold variable-pointer pairs, instead of variable-value pairs. In addition, we add the notion of *storage* which keeps pointer-value pairs. That way, duplicating environment entries only implies copying pointers. Similarly, variable lookups turn into pointer replications.

However, with the introduction of sharing, we also introduce the problem of garbage collection. From the abstract semantics standpoint, this asks the question: "When is the last reference to an entry in the storage gone?" Since we want to enable update-in-place for our arrays, one of the well-established techniques to do so efficiently is a non-delayed garbage collection, also known as reference counting (Hudak, 1986). The fundamental idea here directly relates to the λ-calculus: on each β-reduction, we conceptually have to create as many copies of the argument as we have free occurrences of the bound variable in the body of the function. The reference count reflects these conceptual copies without actually performing them. Whenever such a delayed copy is being consumed, whether by the application of an abstraction or of a primitive function to the delayed copy, its value is conceptually consumed, *i.e.* the reference count is decreased by one. When the reference count drops to zero, the value is discarded from the storage.

To reflect the reference counting scheme in our formalism, we associate a reference count with every pointer in the storage. The *storage* is an ordered list of pointer-value mappings with a *right to left* lookup. Every pointer includes a number of references, which never becomes negative during evaluation. The environment ρ maintains mappings from variables into pointers. Formally, we denote it as follows:

$$S ::= \emptyset \mid S, p \xmapsto{\text{rc}\,(n)} v \qquad \rho ::= \emptyset \mid \rho, x \mapsto p$$

where $p \xmapsto{\text{rc}\,(n)} v$ means that a pointer p binds to a value v and n is its reference count. We look up storages and environments using the notations $S(p)$ and $\rho(x)$, respectively. To look up the number of references of pointer p in a storage S, we use $\text{RC}\,(p,S)$.

We also define a utility operation on storages that makes it possible to alter the reference count value of a particular pointer. We adopt the notation from Chapter 1 of (Pierce, 2004) and write $S \overset{n}{\sim} p$ where p is a binding in S and n is a number by which we want to increase or decrease the reference count of p in S. This means that n can be negative (decrease) or positive (increase). Assuming that p binds to a constant like this: $S = S_l, p \xmapsto{\text{rc}\,(m)} c, S_r$ we have:

$$S \overset{n}{\sim} p = \begin{cases} S \setminus p & m+n=0 \\ S_l, p \xmapsto{\text{rc}\,(m+n)} c, S_r & m+n>0 \end{cases}$$

In case p binds to a closure: $S = S_l, p \xmapsto{\text{rc}\,(m)} [\lambda x.e, \rho], S_r$, we recursively adjust the reference counts of the pointers contained in the environment of the closure as well. Formally, we obtain:

$$S \overset{n}{\sim} p = \begin{cases} S_w & m+n=0 \\ S_l, p \xmapsto{\text{rc}\,(m+n)} [\lambda x.e, \rho], S_r & m+n>0 \end{cases}$$

where:

$$\begin{array}{lcl} \rho & = & x_1 \mapsto p_1, \ldots, x_w \mapsto p_w \\ S_0 & = & S \setminus p \\ S_{k+1} & = & \begin{cases} S_k & p_k \notin S_k \\ S_k \overset{-\#\text{FV}(\lambda x.e, x_k)}{\sim} p_k & p_k \in S_k \end{cases} \end{array}$$

Note that S_i is defined inductively, with the base S_0, and an inductive step that computes S_{k+1} from S_k. The term $\#\text{FV}(e,x)$ refers to the number of free occurrences of x in e.

The new rules have the following judgement:

$$S; \rho \vdash t \Downarrow S'; p$$

where S and ρ are the storage and environment in which the expression e will be evaluated; S' is potentially modified storage after e has been evaluated to p, which is a pointer that can be found in S'. The values that are stored in storages are:

$$v ::= c \mid [\lambda x.e, \rho]$$

We now present the new set of rules. Per convention, we prefix the rules of the storage-based semantics with S-.

$$\textsc{S-Const}\quad \frac{}{S;\rho \vdash c \Downarrow S, p \xmapsto{\text{rc}\,(1)} c;\, p} \qquad\qquad \textsc{S-Var}\quad \frac{x \in \rho \qquad \rho(x) \in S}{S;\rho \vdash x \Downarrow S;\, \rho(x)}$$

Constants enter the storage with reference count one; variables return the pointer that a variable binds to, leaving the storage unmodified.

$$\textsc{S-Sel}\quad \frac{\begin{array}{c} S;\rho \vdash i \Downarrow S_1;\, p_i \qquad S_1(p_i) = v \qquad S_1;\rho \vdash e \Downarrow S_2;\, p_e \\ S_2(p_e) = \{i_1 \mapsto v_1, \ldots, i_n \mapsto v_n\} \qquad \exists k \in \{1 \ldots n\} : i_k = v \qquad S' = S_2 \overset{-1}{\sim} p_i \overset{-1}{\sim} p_e \end{array}}{S;\rho \vdash e\ i \Downarrow S', p \xmapsto{\text{rc}\,(1)} v_k;\, p}$$

$$\textsc{S-Prf}\quad \frac{\begin{array}{c} \oplus \in \{+, -, \ldots\} \qquad S;\rho \vdash e_1 \Downarrow S_1;\, p_1 \\ S_1;\rho \vdash e_2 \Downarrow S_2;\, p_2 \qquad S_2(p_1)\,\text{sem}(\oplus)\,S_2(p_2) = v \qquad S' = S_2 \overset{-1}{\sim} p_1 \overset{-1}{\sim} p_2 \end{array}}{S;\rho \vdash e_1 \oplus e_2 \Downarrow S', p \xmapsto{\text{rc}\,(1)} v;\, p}$$

The pointers are "consumed" in function applications, in the sense that, when a pointer is consumed, its reference count decreases by one. Primitive operations and selections are, in essence, function applications, but with an opaque function body. Therefore, both operations decrease the reference count of each of their arguments by one.

$$\textsc{S-Abs}\quad \frac{}{S;\rho \vdash \lambda x.e \Downarrow S, p_f \xmapsto{\text{rc}\,(1)} [\lambda x.e, \rho];\, p_f}$$

$$\textsc{S-App}\quad \frac{\begin{array}{c} S;\rho \vdash e_1 \Downarrow S_1;\, p_1 \qquad S_1;\rho \vdash e_2 \Downarrow S_2;\, p_2 \qquad S_2(p_1) = [\lambda x.e, \rho_1] \\ \text{FV}(\lambda x.e) = x'_1, \ldots, x'_n \qquad S_3 = S_2 \overset{\#\text{FV}(\lambda x.e, x'_1)}{\sim} \rho(x'_1) \cdots \overset{\#\text{FV}(\lambda x.e, x'_n)}{\sim} \rho(x'_n) \\ S_3 \overset{\#\text{FV}(e,x)-1}{\sim} p_2 \overset{-1}{\sim} p_1;\rho_1, x \mapsto p_2 \vdash e \Downarrow S_4;\, p_3 \end{array}}{S;\rho \vdash e_1\ e_2 \Downarrow S_4;\, p_3}$$

Abstraction extends the storage with a pointer-closure binding with a reference count of one. In the App rule, the reference count of every variable that occurs free in the body of the function is increased by the number of its occurrences (the S_3 storage). This is needed to handle recursive function applications, as we do not know in advance how many times the function will be called. The argument p_2 is ready to be "consumed" at least once, otherwise it would not be in the storage. As we know that there will be $\#\text{FV}(e,x)$ potential usages of p_2, we increase the reference count by $\#\text{FV}(e,x) - 1$. When the argument of a function is not used in the function body, as in the case of $\lambda x.3$, the reference count of p_2 will be simply decreased by one. Finally, as the p_1 binding has been used, we decrease its reference count be one.

The APP rule depends on the following invariant: every closure in the environment holds one copy of its free variables, that is every pointer bound to a free variable x' of the $[\![\lambda x.e,\rho]\!]$ closure has reference count at least $\#\mathrm{FV}(x',\lambda x.e)$. This is true because a variable can become free only by means of previous application or a *letrec*. Prior to application, the reference count of all free variables of a function is increased to the corresponding #FV; when a function binding is discarded from storage, all the free variables are discarded as well.

COND-TRUE
$$\frac{\begin{array}{c}S;\rho\vdash e_1 \Downarrow S_1;\,p \qquad S_1(p)=\mathit{true} \qquad \mathrm{FV}(e_3)=x'_1,\dots,x'_n\\ S_2=S_1\overset{-1}{\sim}p\overset{-\#\mathrm{FV}(e_3,x'_1)}{\sim}\rho(x'_1)\cdots\overset{-\#\mathrm{FV}(e_3,x'_n)}{\sim}\rho(x'_n) \qquad S_2;\rho\vdash e_2\Downarrow S_3;\,p_1\end{array}}{S;\rho\vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3\Downarrow S_3;\,p_1}$$

COND-FALSE
$$\frac{\begin{array}{c}S;\rho\vdash e_1 \Downarrow S_1;\,p \qquad S_1(p)=\mathit{false} \qquad \mathrm{FV}(e_2)=x'_1,\dots,x'_n\\ S_2=S_1\overset{-1}{\sim}p\overset{-\#\mathrm{FV}(e_2,x'_1)}{\sim}\rho(x'_1)\cdots\overset{-\#\mathrm{FV}(e_2,x'_n)}{\sim}\rho(x'_n) \qquad S_2;\rho\vdash e_3\Downarrow S_3;\,p_1\end{array}}{S;\rho\vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3\Downarrow S_3;\,p_1}$$

If we got to evaluation of a conditional with branches e_1 and e_2, the reference count of every free variable x' contained in e_1 and e_2 is set to $\#\mathrm{FV}(x',e_1)+\#\mathrm{FV}(x',e_2)$. This means that when the predicate has been evaluated, we know which branch will be left unevaluated. This means that the reference count of free variables in such a branch has to decrease by the corresponding #FV.

LETREC
$$\frac{\begin{array}{c}S,p\overset{\mathrm{rc}\,(1)}{\longmapsto}\bot;\rho,x\mapsto p\vdash e_1\Downarrow S_1;\,p_1\\ S_2=S_1[p_1/p]_{\mathrm{e}}\setminus p \qquad S_3=S_2\overset{\#\mathrm{FV}(e_2,x)-1}{\sim}p_1 \qquad S_3;\rho,x\mapsto p_1\vdash e_2\Downarrow S_4;\,p_2\end{array}}{S;\rho\vdash \mathit{letrec}\ x=e_1\ \mathit{in}\ e_2\Downarrow S_4;\,p_2}$$

Evaluation of the letrec rule starts with "inventing" a pointer p that binds to the $\bot$ value, which makes the evaluation to diverge, *e.g.* in cases like *letrec* $x = x$ *in* .... We evaluate e_1 to p_1 and we use $[p_1/p]_{\mathrm{e}}$ operation to substitute $\_\mapsto p$ with $\_\mapsto p_1$ in any closure. For example, if *letrec* binds to a recursive function, *e.g.* *letrecf* $=\lambda x.E[f]$ *in* ..., S_2 will contain the following cycle: $p_1\overset{\mathrm{rc}\,(\leq 1)}{\longmapsto}[\![\lambda x.E[f],\dots,f\mapsto p_1,\dots]\!]$.

Our first attempt to define the S-IMAP rule looks like this:

S-IMAP-SEQ

$$\frac{\begin{array}{c} S;\rho \vdash a \Downarrow S_1;\, p_a \\ S_1(p_a) = \{i_1 \mapsto v_1, \ldots, i_n \mapsto v_n\} \qquad S_1;\rho \vdash f \Downarrow S_2;\, p_f \qquad S_2(p_f) = [\lambda i.e, \rho_1] \\ S'_1 = S_2 \overset{|a|-1}{\sim} p_f \qquad \forall k \in \{1,\ldots,n\} : S'_k;\rho, x_f \mapsto p_f \vdash x_f\ i_k \Downarrow S'_{k+1};\, p_k \\ v'_k = S'_{n+1}(p_k) \qquad S^* = S'_{n+1} \overset{-1}{\sim} p_1 \ldots \overset{-1}{\sim} p_n, p \xmapsto{\text{rc}\,(1)} \{i_1 \mapsto v'_1, \ldots, i_n \mapsto v'_n\} \end{array}}{S;\rho \vdash \mathit{imap}\ f\ a \Downarrow S^* \overset{-1}{\sim} p_a;\, p}$$

After evaluating the map function, we increase its reference count by $|a|-1$, to reflect the $|a|$-fold replication of the application of f that the abstract semantics prescribes in the IMAP-rule. The introduction of the fresh variable x_f here is of a purely technical nature to ensure consistency of the formalism.

In the preconditions of the IMAP-rule from the abstract semantics, all function applications of f are independent, *i.e.* the judgements have no interdependencies and, thus, may be executed in any order. In our current S-IMAP-SEQ rule, we unfortunately have defined a fixed order of evaluation of the pre-conditions by passing the modified storage from S'_k to S'_{k+1}. By doing so, we lose one of the key properties of imap — the non-deterministic evaluation order of the function applications, which will make it difficult to reason about parallel execution of the construct. To regain this property, we have to prove that the only possible storage modification during such a function application is in the addition of the result, *i.e.* we have to show that $S_{k+1} \setminus S_k = p_k \xmapsto{\text{rc}\,(1)} v_k$.

3.1 Map/reduce with non-deterministic order

To prove this fact, we formulate the following lemma:

Lemma 1. *If an application of a closure to a constant results in a (potentially different) constant, the only observable change in the storage is the decrement of the reference count of the closure as well as the storage of the result of the application.*

$$\frac{S;\rho \vdash x_f\ c_1 \Downarrow S';\, p \qquad S'(p) = c_2}{S' \overset{-1}{\sim} p = S \overset{-1}{\sim} \rho(x_f)}$$

Proof sketch. The returned value can either be a new value that is not yet in S or it can be a value that is referenced in the environment of the closure behind x_f. If it is a new value, then it can only be a constant or the result of a primitive function. In that case, all relatively free variables in the body of the function behind x_f have to be consumed as often as they occur in the body, *i.e.* we have $S' = S \overset{-1}{\sim} \rho(x_f), p \xmapsto{\text{rc}\,(1)} c_2$, which is equivalent to our goal. If the returned value already exists in S, it has to be within the environment bound in the closure. In that case, the corresponding variable is the only one that is *not* consumed in the body of the closure, *i.e.* we have $S' = S \overset{1}{\sim} p \overset{-1}{\sim} \rho(x_f)$ which is also equivalent to our goal expression. □

Using Lemma 1 we can observe for the preconditions in the S-IMAP-SEQ-rule that:

$$\forall k \in \{1,\ldots,n\} : S;\rho, x_f \mapsto p_f \vdash x_f\ i_k \Downarrow S^k;\, p_k \qquad \text{with} \qquad S^k \overset{-1}{\sim} p_k = S \overset{-1}{\sim} p_f$$

which means that we can perform the iterations of the imap-operator in any arbitrary order, as all the p_k-s will be discarded from the storage at the end of the *imap* evaluation. This results in the same storage that we started with, apart from the consumed arguments of the *imap*-construct and the generated overall result.

Finally, we observe that there is no need to to accumulate all the intermediate results and then to collect the values into an array. Instead we can preallocate the resulting array at the beginning, and update the corresponding elements at every iteration of the *imap*.

To define the new rules for *imap* and *reduce* we will use the idea of pretty-big-step operational semantics (Charguéraud, 2013) and introduce some auxiliary terms to avoid conditions in the rules. The new terms are:

imap$_1$ $p_1\ I\ p_2$ — the term that captures the pointer p_1 where the result of the evaluated *imap* will be stored, as well as the iteration space I which is a set of indices. The last expression p_2 is the pointer that binds to a closure of the *imap*'s inner function.

reduce$_1$ $p_1\ I\ p_2\ p_3$ — the same idea as with *imap*$_1$, however for *reduce* we also need to pass the pointer to the argument array, which we do via p_3.

Here are the final rules for *imap* and *reduce*; we use $\perp$ to denote an undefined value. The notation $S \oplus_p v$ means that we update the binding p in S to the new value v, preserving its original reference count.

S-IMAP
$$\frac{\begin{array}{c} S;\rho \vdash a \Downarrow S_1;\, p_a \qquad S_1(p_a) = \{i_1 \mapsto v_1, \ldots, i_n \mapsto v_n\} \\ S_1;\rho \vdash f \Downarrow S_2;\, p_f \qquad S_3 = S_2 \overset{|a|-1}{\sim} p_f, p \xmapsto{\mathrm{rc}\,(1)} \{i_1 \mapsto \perp, \ldots, i_n \mapsto \perp\} \\ I = \{i_1, \ldots, i_n\} \qquad S_3;\rho \vdash \mathit{imap}_1\ p\ I\ p_f \Downarrow S_4;\, p \end{array}}{S;\rho \vdash \mathit{imap}\ f\ a \Downarrow S_4 \overset{-1}{\sim} p_a; p}$$

S-IMAP-1.1
$$\frac{}{S;\rho \vdash \mathit{imap}_1\ p\ \emptyset\ p_f \Downarrow S;\, p}$$

S-IMAP-1.2
$$\frac{\begin{array}{c} i_k \in I \qquad S(p) = \{i_1 \mapsto v_1, \ldots, i_k \mapsto \perp, \ldots, i_n \mapsto v_n\} \qquad S;\rho, x_f \mapsto p_f \vdash x_f\ i_k \Downarrow S';\, p_k \\ S_1 = S \overset{-1}{\sim} p_f \oplus_p \{i_1 \mapsto v_1, \ldots, i_k \mapsto S'(p_k), \ldots, i_n \mapsto v_n\} \\ S_1;\rho \vdash \mathit{imap}_1\ p\ I \setminus \{i_k\}\ p_f \Downarrow S_2;\, p \end{array}}{S;\rho \vdash \mathit{imap}_1\ p\ I\ p_f \Downarrow S_2;\, p}$$

We will never read a value $\perp$, because p has no direct references outside the *imap* context, and the iteration space I is identical to the number of $\perp$ in p. This means that, after the *imap* is evaluated, all the $\perp$ values will have been be replaced by new values.

By creating an explicit iteration space, we ensure that iterations can happen in arbitrary order and that the result of each function application is discarded at every iteration. We need not pass the argument array explicitly, because the inner function is applied to the indices. If the array is referenced inside the function, it will be a free variable of that function.

The same idea can be applied to the reduce operator, as that its inner function is associative and commutative.

S-REDUCE

$$\frac{S;\rho \vdash a \Downarrow S_1;\, p_a \qquad S_1(p_a) = \{i_1 \mapsto v_1, \dots, i_n \mapsto v_n\} \qquad S_1;\rho \vdash f \Downarrow S_2;\, p_f \\ S_3 = S_2 \overset{|a|-1}{\sim} p_f, p \xmapsto{\text{rc}\,(1)} e_{\text{neut}} \qquad I = \{i_1, \dots, i_n\} \qquad S_3;\rho \vdash \mathit{reduce}_1\ p\ I\ p_f\ p_a \Downarrow S_4;\, p}{S;\rho \vdash \mathit{reduce}\ f\ a \Downarrow S_4 \overset{-1}{\sim} p_a; p}$$

S-REDUCE-1.1

$$\frac{}{S;\rho \vdash \mathit{reduce}_1\ p\ \emptyset\ p_f\ p_a \Downarrow S;\, p}$$

S-REDUCE-1.2

$$\frac{i_k \in I \qquad S(p) = v \\ S(p_a) = \{i_1 \mapsto v_1, \dots, i_k \mapsto v_k, \dots, i_n \mapsto v_n\} \qquad S;\rho, x_f \mapsto p_f \vdash x_f\ v\ v_k \Downarrow S';\, p_k \\ S_1 = S \overset{-1}{\sim} p_f \oplus_p S'(p_k) \qquad S_1;\rho \vdash \mathit{reduce}_1\ p\ I \setminus \{i_k\}\ p_f\ p_a \Downarrow S_2;\, p}{S;\rho \vdash \mathit{reduce}_1\ p\ I\ p_f\ p_a \Downarrow S_2;\, p}$$

Although empty arrays are not supported by our type system, the map/reduce rules above are capable of dealing with them. In case of *imap*, the allocated result would be empty, I would be empty as well, which means that we apply S-IMAP-1.1 and get the result. In case of *reduce*, this is because we bind the e_{neut} value to the pointer that will be returned as result. Hence, a reduction over an empty array produces a neutral element.

Semantic equivalence To demonstrate that the new semantics evaluate the same values, we first have to show that its progress is not affected by our introduced reference-counting operations.

Theorem 1. *Evaluation will never get stuck because of a failed pointer look-up. Given a derivation tree of the form:*

$$\frac{\begin{array}{c} S_1;\rho \vdash e_1 \Downarrow S_2;\, p_1 \\ \dots \\ S_k;\rho \vdash e_k \Downarrow S_{k+1};\, p_k \\ \dots \\ S_n;\rho \vdash e_n \Downarrow S_{n+1};\, p_n \end{array}}{S_1;\rho \vdash e \Downarrow S_{n+1};\, p_n}$$

and given that

$$p \in S_k \wedge p \notin S_{k+1} \implies e_{k+1}, \dots, e_n \text{ do not consume } p$$

Proof sketch. This follows from the isomorphism of evaluation under current semantics and semantics with substitutions (not presented in this paper). The essence of the reference counting lies in delaying a copy that happens during the substitution. During evaluation of the $\lambda x.e\ c$ expression, x is substituted with c in e #FV(x,e) times. This means that to evaluate e we have to maintain #FV(x,e) copies of c. However, we have one copy of c already that comes from the argument, therefore at every function application we need #FV$(x,e) - 1$ copies of the argument. If an argument is a closure that captures free variables, then during the application of the enclosed function, the enclosed free variables will be used as many times as the function will be used. That is where the recursive nature of $S \overset{n}{\sim} p$ comes from. Now, when a variable is being looked up, it will be either consumed in another function application, or returned as a result. Therefore, there is no other way to access a pointer that binds to a variable in the environment, as after the body of a function is evaluated, the environment does not exist anymore. As it follows, from abstract semantics, maps and reduce encapsulate n function applications, that is why the reference count has to be adjusted accordingly. $\square$

Once we have the above theorem, we can formulate a semantic equivalence theorem.

Theorem 2. *Abstract semantics evaluates the same values as semantics with storages:*

$$\emptyset \vdash e \Downarrow_A v \wedge \emptyset; \emptyset e \vdash \Downarrow_S S;\, p \implies v = S(p)$$

where $\Downarrow_A$ and $\Downarrow_S$ refers to evaluation using abstract and storage-based semantics respectively.

Proof sketch. If we remove all $S \overset{n}{\sim} p$ operations from the current rules, we obtain semantics in which values are never discarded from the storage. The isomorphism between this semantics and abstract semantics is straightforward, because the only difference between them is the order of evaluation of pre-conditions. Environments remain the same in both cases, with respect to variable names and order. After that, we run induction on the rules of semantics without reference counting and with reference counting, and the only thing we need to show is that discarding pointers does not impact progress of the evaluation. This fact follows from Theorem 1. $\square$

Example program evaluation We consider now an example program evaluation to develop an intuition of how the rules work, specifically when we deal with λ-abstractions with closures and function applications.

$$\begin{array}{lrr}
\emptyset; \emptyset & (\lambda f.f\ 2)\ (\lambda x.(\lambda y.x)\ 3) & \textsc{Abs} \\
p_f \overset{\mathrm{r}(1)}{\mapsto} [\lambda f.f\ 2, \emptyset]; \emptyset & p_f\ (\lambda x.(\lambda y.x)\ 3) & \textsc{Abs} \\
S = p_f \overset{\mathrm{r}(1)}{\mapsto} [\lambda f.f\ 2, \emptyset], p_x \overset{\mathrm{r}(1)}{\mapsto} [\lambda x.\lambda y.x, \emptyset]; \emptyset & p_f\ (p_x\ 3) & \textsc{Const} \\
S, p_3 \overset{\mathrm{r}(1)}{\mapsto} 3; \emptyset & p_f\ (p_x\ p_3) & \textsc{App}[p_x] \rightarrow \\
S_1 = S, p_3 \overset{\mathrm{r}(1)}{\mapsto} 3 \overset{0}{\sim} p_3; x \mapsto p_3 & \lambda y.x & \textsc{Abs} \\
S_2 = S_1 \ominus p_x, p_y \overset{\mathrm{r}(1)}{\mapsto} [\lambda y.x, x \mapsto p_3]; x \mapsto p_3 & p_y & \leftarrow \\
S_2; \emptyset & p_f\ p_y & \textsc{App}[p_f] \rightarrow \\
S_2 \overset{0}{\sim} p_y; f \mapsto p_y & f\ 2 & \textsc{Var} \\
S_2; f \mapsto p_y & p_y\ 2 & \textsc{Const} \\
S_2, p_2 \overset{\mathrm{r}(1)}{\mapsto} 2; f \mapsto p_y & p_y\ p_2 & \textsc{App}[p_y] \rightarrow \\
S_2, p_2 \overset{\mathrm{r}(1)}{\mapsto} 2 \overset{-1}{\sim} p_2; x \mapsto p_3 & x & \textsc{Var} \\
S_2 \ominus p_y; x \mapsto p_3 & p_3 & \leftarrow \\
S_2 \ominus p_f; \emptyset & p_3 & \leftarrow \\
p_3 \overset{\mathrm{r}(1)}{\mapsto} 3; \emptyset & p_3 & \square
\end{array}$$

A higher-order function $\lambda f.f\ 2$ is applied to the function that encloses variable x. The evaluation starts by building closures for the $\lambda f.f\ 2$ and $\lambda x.\lambda y.x$ functions. After that, the value 3 is mapped to the pointer p_3 and added to the storage. Application of p_x to p_3 creates a new environment, which we denote using '$\text{APP}[p] \rightarrow$', where p is a pointer to the function. When the last environment is destroyed, we use '$\leftarrow$'. The application of p_x to p_3 does not change the reference count of the p_3, as there is only one occurrence of x in the body of the p_x function. To evaluate the abstraction $\lambda y.x$, we have to create a closure that encloses current binding of x. This closure is mapped to the pointer p_y and is added to the storage. When p_y is passed as an argument, its reference count is altered by zero, which means that the reference count of p_y and the enclosed pointer p_3 did not change. At the application of p_y to the value 2, the reference count of the pointer that stores the value 2 is immediately decreased by one, making it zero and removing it from the environment. Finally, all the closures are discarded from the storage because of $\ominus$ operations, and p_3 is returned as result.

3.2 Reuse criteria

In this subsection, we use Hudak's ideas on aggregate updates (Hudak & Bloss, 1985) in application to λ_{DP} and sketch static analysis and extension of the semantics that are needed.

Conceptually, the semantics with storages presented thus far require that every *map* and *imap* to allocate a new pointer for the result and to update elements of the newly allocated array at every index. Although this is operationally correct, we can often exploit an optimisation that significantly improves performance. Consider the case when only one element is updated in a large array. In an imperative language like C one might write:

```
a[42] += 1;
```

In λ_{DP}, any modification of an array is achievable only by means of map operations. Given that we have conditions in λ_{DP}, the former example could be expressed as follows:

$$\mathit{imap}\ \lambda i.\,(\mathit{if}\ i = 42\ \mathit{then}\ (a\ i) + 1\ \mathit{else}\ (a\ i))\ \ a$$

Note a subtle difference between C and λ_{DP}. In λ_{DP}, the operation will create a new array, where all the elements except the one at index 42 will be *copied* into the newly allocated memory. The value of the element at index 42 will be recomputed. Given that the array a has reference count one, this creates a huge overhead.

To avoid such copying, one has to use a instead of the newly allocated memory and, when evaluating *imap*, eliminate copy operations. In application to our example, it means that instead of:

$$\mathit{imap}_1\ p\ \{i_1, \dots, i_n\}\ p_f$$

we would like to get:

$$\mathit{imap}_1\ p_a\ \{42\}\ p_f$$

assuming that

$$S(p_f) = \lambda i.\mathit{if}\ i = 42\ \mathit{then}\ (a\ i) + 1\ \mathit{else}\ (a\ i) \qquad\qquad \rho(a) = p_a$$

in current S and ρ.

In order to enable such an optimisation, one has to prove that it is safe to replace p with p_a and then to compute the set of indices, for which the *imap* function computes the identity. First, we introduce the reuse criteria for the $imap_1\ p\ I\ p_f$ operation which makes it possible to substitute p with some p_x. Assume that:

$$S(p_f) = [f \equiv \lambda i.e, \rho_1] \qquad\qquad \rho(x) = p_x$$

We can reuse arrays that will be discarded at the end of the *imap* operation. This means that it can either be an argument array of the *imap*, or it is one of the free variables in the body of the *imap*'s function. We ignore the case when we reuse the argument array, as in the implementation, the *imap*'s second argument is often presented in a compact form, which means that it is never materialised in memory. For the arrays that appear free in the *imap*'s function, we have to make sure that:

$$x \in \mathrm{FV}(f) \wedge \mathrm{RC}(p_x) = \#\mathrm{FV}(f, x)$$

The reused array has to be of the same shape, and have the same set of indices:

$$I(x) = I(a) = I$$

After that, we have to inspect all selections on the array that we are about to reuse. For that we have to assume that we can bring f to the following normal form:

$$f\ j = F\ (x\ (g_1\ j))\ \ldots (x\ (g_n\ j)) \wedge \#\mathrm{FV}(F, x) = 0$$

that is, to lift all the selection operations on the array x.

Definition 1. *The array bound to the variable x can be reused in the imap with inner function f and iteration space I: denoted $R(f, I, x)$; if every new element of the result at index j depends on either a copy of the non-modified element of x or on the element at index j.*
$\forall j \in I : \forall i \in \{g_1\ j, \ldots, g_n\ j\}$*:*

$$i = j \tag{R.1}$$

or

$$(f\ i) = (x\ i) \tag{R.2}$$

To reflect this in the semantics, we adjust the S-IMAP rule:

S-IMAP-R

$$\frac{\begin{array}{c} S;\rho \vdash a \Downarrow S_1;\ p_a \qquad S_1(p_a) = \{i_1 \mapsto v_1, \ldots, i_n \mapsto v_n\} \qquad S_1;\rho \vdash f \Downarrow S_2;\ p_f \\ I = \{i_1, \ldots, i_n\} \qquad S_2(p_f) = [f' \equiv \lambda i.e, \rho_1] \qquad x \in \mathrm{FV}(f') \qquad I(x) = I(a) \\ \mathrm{RC}\,(\rho_1(x), S_2) = \#\mathrm{FV}(f', x) \qquad R(f', I, x) \qquad I' = \{j, f\ j \neq x\ j \mid j \in I\} \\ S_3 = S_2 \overset{|I'|-1}{\sim} p_f \overset{1}{\sim} \rho_1(x) \qquad S_3;\rho \vdash imap_1\ \rho_1(x)\ I'\ p_f \Downarrow S_4;\ p \end{array}}{S;\rho \vdash imap\ f\ a \Downarrow S_4 \overset{-1}{\sim} p_a;\ p}$$

The parts of the rule that are new or modified are marked in red. We check that the variable appears free in the *map*'s function; that it will be discarded from the environment after *imap*'s evaluation and that the *imap* adheres to the reuse criteria for every index of the argument array. If this holds, we construct the iteration space I' which contains only those indices of I where the *imap*'s inner function does not act as identity function applied to the corresponding element of the array x. Finally we increase the number of references to the function (and the enclosed free variables by transitivity of $\overset{n}{\sim}$) so that f' could be evaluated

$|I'|$ times, then increase the number of references of the resulting array by one, so that it will not be discarded from the storage.

This is the only change that we need to make to add destructive updates to λ_{DP}. The sub-rules S-IMAP-1.1 and S-IMAP-1.2 can be applied with no further changes.

Lemma 2. *Storage-based semantics with reuses evaluates the same values as the storage-based semantics.*

$$\frac{S;\rho \vdash e \Downarrow_{\text{S-IMAP}} S_1;p_1 \qquad S;\rho \vdash e \Downarrow_{\text{S-IMAP-R}} S_2,p_2}{S_1(p_1) = S_2(p_2)}$$

Proof sketch. Induction on iterations of the *imap* using the fact that $R(f,I,x)$ holds. □

3.3 Trade-offs in λ_{DP}

The overall presentation of λ_{DP} and its semantics described so far comes with a number of restrictions and shortcuts. Some of them are there "by design"; others are discussed within the rest of the subsection.

Observations The concept of storage and environment maps very well to the real systems, as storage is a model of heap and environment is a model of stack.

When compiling λ_{DP}, we can generate explicit statements to increment/decrement the reference count of a pointer. This would be faster than looking up the reference counter for pointers.

To simplify the analysis required for the reuse criteria, we can extend the *imap* syntax, at least to capture the idea that only a certain subspace of the argument array is being updated.

Practical extensions Restricting array types simplifies our reference counting scheme. Nothing stops us from allowing nested arrays or arrays of functions. All we have to do is to propagate reference counting in the nested structure, akin to the way it propagates to the enclosed pointers of a closure.

To support conditionals in λ_{DP}, the reference counting during evaluation and free-variable counting have to be extended. When counting free variables of a conditional, we have to take a union of both branches. When counting a number of references for a given variable, we would need to take a sum of number of occurrences in both branches. At runtime, when a predicate is to be evaluated and we know which branch we execute, for every variable that occurs in both branches we need to subtract the number of occurrences in the branch that we did not execute.

Recursive functions are possible, but require extra care. If we follow Kahn's natural semantics, recursive functions are introduced as closures which enclose an environment with self-references. This will break our reference counting scheme, as the $S \overset{n}{\sim} p$ operation might never terminate. If we introduce recursion via a fixed-point combinator, we have to get a new term in the language for it, extend the type system so that programs with fixpoints are typeable. That will be sufficient, as recursive functions that on will always appear as parameters, and the reference counting will be automatically updated on every unfolding of the fix combinator.

Implementation concerns Another way of thinking about the reuse property is in terms of dependency analysis. We have to guarantee that at every iteration there will be no read-after-write dependencies. There exists a large body of work that explains how to perform such an analysis. Most noticeably (Kennedy & Allen, 2002; Allen *et al.*, 1983) and the work that uses the polyhedral model (Feautrier, 1991; Pouchet *et al.*, 2007). Building exact formalisms on identify reuse in λ_{DP} is beyond the scope of this paper. However, it is of significant importance that, given R and the procedure on how to construct I', destructive updates can be supported by λ_{DP} semantics.

4 Shared memory semantics

We now present shared-memory semantics of λ_{DP}, and demonstrate that it evaluates the same values as the abstract semantics.

4.1 Concurrency in λ_{DP}

The *map*, *imap* and *reduce* constructs provide a natural source of concurrency that can be exploited by implementations of λ_{DP}: concurrency opportunities appear as independent preconditions in the rules of the operational semantics. The appeal here is that concurrency can be exploited immediately, without any further analysis or proofs. Alternatively, one needs to analyse the unfolding of the evaluation tree to show that for certain terms their parts can be evaluated concurrently under some conditions.

In λ_{DP}, map/reduce are not the only possible sources of concurrency. The abstract semantics of App, Sel, and Prf also have independent preconditions. We sequentialise those in the semantics with storages, but it would be easy to show that they can be evaluated in arbitrary order. Nevertheless, we consider that all three operations are evaluated sequentially. The justification for sequential evaluation is that, by allowing such parallelism, the implementation would have to deal with the tree of parallel tasks, where each task is very fine-grained. The problem with this is that number of synchronisations required to fold the tree back gets very large, which can often result in decreased performance. In this paper, we only consider non-nested parallelism that implements map/reduce operations. That is, if we choose to evaluate a map/reduce in parallel, all its sub-expressions must be evaluated sequentially. However, bear in mind that this model extends to nested data-parallelism using Blelloch's flattening (Blelloch & Sabot, 1988).

4.2 Distributions

When implementing map/reduce operations, one has to decide how to schedule iterations among the theoretically available number of threads. Further in the paper we enumerate threads from 1 to $t_{\max}$. To capture the idea of scheduling, we introduce the notion of *distribution*.

A distribution prescribes how the index space of the given array is divided among $t_{\max}$ threads. Formally, we extend the notion of arrays as follows:

$$c ::= 0, 1, \dots \mid \langle d, \{c \mapsto c, \dots, c \mapsto c\} \rangle \qquad\qquad d ::= \star \mid D_a$$

An array is now a two-element tuple, where the first element is a distribution, and the second element is the mapping of indices to values as before. To access the elements of the tuple, we introduce meta-operators $\underline{\text{fst}}$ and $\underline{\text{snd}}$, which select the first and the second element of the tuple, respectively.

A distribution is a mapping from thread numbers to sub-index-spaces, defined for every $t \in \{1,\dots,t_{\max}\}$. For example, for an array

$$a = \langle D_a, \{i_1 \mapsto a_1, \dots, i_n \mapsto a_n\}\rangle$$

the distribution D_a is structured as follows:

$$D_a = \{1 \mapsto \{i_1^1, \dots\}, \dots, t_{\max} \mapsto \{i_1^{t_{\max}}, \dots\}\}$$

To look up the distribution, we use the meta-operator @. For example, assuming that $t_{\max} = 3$ and

$$D_a = \{1 \mapsto \{0\}, 2 \mapsto \{\}, 3 \mapsto \{1,2\}\}$$

the @ operator will return:

$$D_a@1 = \{0\} \qquad D_a@2 = \{\} \qquad D_a@3 = \{1,2\}$$

Note that for certain threads, the index space can be empty. We also introduce a special kind of distribution, which is denoted as $\star$. This means that all the elements of the array will reside on a single thread, without specifying which thread. In a sense, this is a polymorphic distribution over the thread numbers.

We require distributions to be complete and unique:

$$\textsc{Completeness}\quad \frac{a = \langle D_a, V_a\rangle}{\bigcup_{t=1}^{t_{\max}} D_a@t = \{i \mid (i \mapsto \_) \in V_a\}} \tag{D.1}$$

$$\textsc{Uniqueness}\quad \frac{a = \langle D_a, V_a\rangle}{\forall (i,j) \in \{1,\dots,t_{\max}\}^2 \wedge i \neq j : D_a@i \cap D_a@j = \emptyset} \tag{D.2}$$

Completeness means that a distribution maps every index of an array to some thread. Uniqueness means that every array index is mapped only to one thread.

Now we define semantics for shared memory that uses distributions to make scheduling decisions.

4.3 Rules for shared-memory semantics

Semantics for shared memory makes the notion of a thread explicit in the evaluation context. The assumption is that every expression is being evaluated within a certain thread t, and that the result is found at the same thread where the evaluation started. The overall number of threads available on the system is fixed to $t_{\max}$ and is an unchanging, global parameter of the evaluation.

The storage S has the same structure as before, but it is now being shared among all threads. To avoid race conditions, we must prevent simultaneous updates of the same pointers from within different threads. Most importantly, we assume that reference count operations $\overset{n}{\sim}$ happen atomically.

To avoid nested unfolding of the tasks when evaluating map/reduce constructs, we introduce a global state $s = \{+, -\}$ that will be "carried around" during the evaluation. The '+' state means that the thread is allowed to create new threads, whereas '-' indicates that we are within a parallel map/reduce operation in which the entire expression must be evaluated locally within t.

The judgement for the shared-memory semantics has the following form:

$$S; t^s; \rho \vdash e \Downarrow S'; p \qquad\qquad s ::= + \mid -$$

Now we define the new set of rules. Per convention, we are going to prefix them with T-. The rules Const, Var, App will not change, and the thread number t will be propagated through the pre-conditions together with its current state s. Consider this at the example of the App rule:

$$\textsc{T-App}\quad \frac{\begin{array}{c} S; t^s; \rho \vdash e_1 \Downarrow S_1;\, p_1 \qquad S_1; t^s; \rho \vdash e_2 \Downarrow S_2;\, p_2 \\ S_2(p_1) = [\lambda x.e, \rho_1] \qquad S_2 \overset{\#\mathrm{FV}(e,x)-1}{\sim} p_2; t^s; \rho_1, x \mapsto p_2 \vdash e \Downarrow S_3;\, p_3 \end{array}}{S; t^s; \rho \vdash e_1\ e_2 \Downarrow S_3 \ominus p_1;\, p_3}$$

Regardless of what the value of s is, we simply propagate it into the pre-conditions.

Selections have to be adopted to the new format of arrays:

$$\textsc{T-Sel}\quad \frac{\begin{array}{c} S; t^s; \rho \vdash i \Downarrow S_1;\, p_i \qquad S_1(p_i) = v \qquad S_1; t^s; \rho \vdash e \Downarrow S_2;\, p_e \\ S_2(p_e) = \langle D_e, \{i_1 \mapsto v_1, \ldots, i_n \mapsto v_n\}\rangle \qquad \exists k \in \{1 \ldots n\} : i_k = v \qquad S' = S_2 \overset{-1}{\sim} p_i \overset{-1}{\sim} p_e \end{array}}{S; t^s; \rho \vdash e\ i \Downarrow S', p \xmapsto{\mathrm{rc}\,(1)} v_k;\, p}$$

Note that the essence of the operation does not change.

The *map*, *imap* and *reduce* rules change substantially, as we have to consider a multi-threaded execution of the construct in the case of t^+, and a sequential one in the case of t^-. The choice which *imap* should be executed in parallel is a difficult problem beyond the scope of this paper. The rules presented for both cases execute an *imap* in parallel whenever the argument has a non-$\star$ distribution and we are in '+' state. In principle, the process of choice could be more sophisticated.

T-IMAP-PAR

$$\begin{array}{c}
S;t^+;\rho \vdash a \Downarrow S_1;\, p_a \qquad S_1(p_a) = \langle D_a, \{i_1 \mapsto v_1, \dots, i_n \mapsto v_n\}\rangle \\
S_1;t^+;\rho \vdash f \Downarrow S_2;\, p_f \qquad S_3 = S_2, p \xrightarrow{\mathrm{rc}\,(1)} \langle D_a, \{i_1 \mapsto \bot, \dots, i_n \mapsto \bot\}\rangle \\
\overset{t_{\max}}{\underset{k=1}{\forall}} : S_3 \overset{|D_a@k|}{\sim} p_f;k^-;\rho \vdash imap_1\ p\ D_a@k\ p_f \Downarrow S^k;\, p \\
S_u = \bigcup_{k=1}^{t_{\max}} S^k \overset{-1}{\sim} p_f \qquad S' = S_u \overset{-1}{\sim} p_a \\
\hline
S;t^+;\rho \vdash imap\ f\ a \Downarrow S';p
\end{array}$$

T-IMAP-SEQ

$$\begin{array}{c}
S;t^-;\rho \vdash a \Downarrow S_1;\, p_a \qquad S_1(p_a) = \langle d, \{i_1 \mapsto v_1, \dots, i_n \mapsto v_n\}\rangle \qquad d = \star \vee s = - \\
S_1;t-;\rho \vdash f \Downarrow S_2;\, p_f \qquad S_3 = S_2 \overset{|a|-1}{\sim} p_f, p \xrightarrow{\mathrm{rc}\,(1)} \langle \star, \{i_1 \mapsto \bot, \dots, i_n \mapsto \bot\}\rangle \\
I = \{i_1, \dots, i_n\} \qquad S_3;t^-;\rho \vdash imap_1\ p\ I\ p_f \Downarrow S_4;\, p \\
\hline
S;t^s;\rho \vdash imap\ f\ a \Downarrow S_4 \overset{-1}{\sim} p_a;p
\end{array}$$

Consider the T-IMAP-PAR rule. We have marked the differences with IMAP in red, except when we have added t^s to preconditions. The fork part is achieved by calling the $imap_1$ with the iteration space $D_a@k$. Note that when we do this we change the thread status to t^-. The join part happens when we need to unify $t_{\max}$ storages that might have diverged into a single one.

Now, by looking at S-IMAP-1.1 and S-IMAP-1.2 we can conclude that S^k contains p with values at $D_a@k$ indices being updated. The completeness of distributions implies that all the elements of p will be updated when all the threads terminate.

When we run $t_{\max}$ instances of the *imap*, we have conceptually copied the S_3 storage to every thread. From Lemma 1, we know that S_3 and S^k will differ only in the value of p. Also, the reference count of p_f has to be decreased by one in every S^k, as it has not been consumed by the *imap*. This means, in practice, that we need not copy storages, as they are shared, so we can increase the reference count of p_f by $|a|$ beforehand. In this case, the union of S^k is simply a barrier that waits until all the threads terminate.

In the case of T-IMAP-SEQ, we observe that it is almost identical to the S-IMAP. Note that T-IMAP-SEQ ignores the distribution of the argument array and evaluates all the values within t. We also change the distribution of the result from D_a to $\star$, indicating that the result is not shared.

For the shared-memory *reduce* rules, we follow the same principle as *imap* and derive the T-REDUCE-PAR rule from the S-REDUCE rule.

T-REDUCE-PAR

$$\frac{\begin{array}{c} S;t^{+};\rho \vdash a \Downarrow S_1;\, p_a \qquad S_1(p_a) = \langle D_a, V_a\rangle \\ S_1;t^{+};\rho \vdash f \Downarrow S_2;\, p_f \qquad S_3 = S_2 \overset{|a|+t_{\max}-1}{\sim} p_f,\, p_1 \xmapsto{\text{rc }(1)} e_{\text{neut}}, \ldots, p_{t_{\max}} \xmapsto{\text{rc }(1)} e_{\text{neut}} \\ \overset{t_{\max}}{\underset{k=1}{\forall}} : S_3;t^{-};\rho \vdash \mathit{reduce}_1\ p_k\ D_a@k\ p_f\ p_a \Downarrow S^k;\, p_k \qquad S_u = \bigcup_{k=1}^{t_{\max}} S^k \\ S_{uu} = S_u, p \xmapsto{\text{rc }(1)} \langle \star, \{1 \mapsto S_u(p_1), \ldots t_{\max} \mapsto S_u(p_{t_{\max}})\}\rangle \\ S_{uu}, p_r \xmapsto{\text{rc }(1)} e_{\text{neut}};t^{-};\rho \vdash \mathit{reduce}_1\ p_r\ I(p)\ p_f\ p \Downarrow S';\, p_r \\ S'' = S' \overset{-1}{\sim} p_1 \ldots \overset{-1}{\sim} p_{t_{\max}} \overset{-1}{\sim} p \overset{-1}{\sim} p_a \end{array}}{S;t^{+};\rho \vdash \mathit{reduce}\ f\ a \Downarrow S'';\, p_r}$$

Assuming that the argument function of the *reduce* is commutative and associative, we can split the iteration space of the argument array according to distributions, then evaluate partial reductions of the each part in p_k. After that we join all threads, assemble partial results into the intermediate array, and run reduce with the same function again. Such a scheme requires $|a| + t_{\max}$ instances of the argument function, which is reflected during the construction of S_3. After the evaluation of the final result, we discard the intermediate array and partial results. We do not provide the rule for T-REDUCE-SEQ, as it can be directly derived from the S-REDUCE rule.

Theorem 3. *Parallel semantics evaluate the same values as semantics with storages.*

$$\frac{\emptyset;\emptyset \vdash e \Downarrow_S S_1;\, p_1 \qquad \emptyset;t^{+};\emptyset \vdash e \Downarrow_T S_2;\, p_2}{S_1(p_1) = S_2(p_2)}$$

Proof sketch. On the similarity of rules. All the rules except map/reduce can be trivially matched. For map/reduce, using completeness and uniqueness of distributions, we show that the number of references in the final storage during the evaluation under shared-memory semantics is the same as after evaluation under storage-based semantics. □

In the above theorem, we state that we start running the program with a thread t without specifying a particular t. This works, because in the presented model, any thread can be considered as the master thread (the one who does forks and joins).

Reuse in the shared-memory semantics Reuse criteria can be still used in the shared-memory semantics, when running an *imap* in parallel; but it requires some adjustments.

T-IMAP-PAR-R

$$\frac{\begin{array}{c} \ldots \\ \overset{t_{\max}}{\underset{k=1}{\forall}} : S_3 \overset{|D_a@k\cap I'|}{\sim} p_f;t^{-};\rho \vdash \mathit{imap}_1\ p\ D_a@k\cap I'\ p_f \Downarrow S^k;\, p \qquad \ldots \end{array}}{S;t^{+};\rho \vdash \mathit{imap}\ f\ a \Downarrow S';\, p}$$

Mainly, if R holds, when computing the index space for a given thread, we intersect I' with the index space of the thread that is coming from the distribution.

Lemma 3. *Reuse criteria holds in parallel semantics:*

$$\frac{t^+;\rho \vdash e \Downarrow_{\text{T-IMAP-PAR}} S_1;p_1 \qquad S;t^+;\rho \vdash e \Downarrow_{\text{T-IMAP-PAR-R}} S_2;p_2}{S_1(p_1) = S_2(p_2)}$$

Proof sketch. Induction on the iterations of the *imap*, using the *R* property and correctness of the T-IMAP-PAR □

Implementation concerns From the S-IMAP-1.2 rule, we notice that at every iteration the storage contains an updated value p and p_f which reference count is decreased by one. This means that all the decrements of the p_f could happen at the end of S-IMAP rule. To achieve this, we have to inhibit the reference at every p_f application. In the multi-threaded context this is a very valuable implementation trick, as every reference counting potentially leads to a lock. Therefore, not doing reference counting during the evaluation of the *imap* leads to substantial performance increase.

5 Semantics for distributed memory

In contrast to the shared-memory system where one thread executes exclusively until the first *imap* on a distributed array is met, the distributed memory system follows an SPMD approach (Single Program Multiple Data), *i.e.* all threads execute the same code until the first *imap* is met. This design has the advantage that all non-distributed entries in the storage are available in *all* local storages avoiding a communication of these values upon startup of parallel executions.

As there is no global memory on a distributed-memory system, we must decide how to share arrays across nodes. From now on, distributions serve a dual purpose, they do not only prescribe which process computes what part of the array but they also define on which node parts of the arrays are stored across the cluster. This implies that all array writes during the evaluation of distributed imap/reduce operations can happen locally on each node. During the synchronisation phase of imap, no data exchange among nodes is needed. All reads to non-local parts of the distributed arrays will happen at runtime, potentially requiring internode communication.

When *imap* or *reduce* is evaluated under the shared memory semantics, conceptually, we copy the storage S_3 to every thread and then obtain the modified storage back. In case of shared memory, the copying can be avoided, as S_3 can be shared amongst the threads. If we replicate the same model on a cluster, we would have to physically copy storages (or relevant part of a storage) to each of the nodes. To avoid this, we replicate local computations, *i.e.* those that are evaluated in t^+ mode, on each node. Given that the compute capabilities of all the nodes are the same, recomputing local parts on each node, instead of sending data from a single node, is a beneficial trade-off as it entirely eliminates storage communication.

To capture such non-local reads formally, we maintain a conceptual copy of *all* local storages at *all* nodes. This makes the new judgements look as follows:

$$\bar{S};t^s;\rho \vdash e \Downarrow \bar{S}';p$$

where $\bar{S}$ and $\bar{S}'$ are tuples of node-local storages:

$$\bar{S} = \langle S_1, \ldots, S_{t_{\max}} \rangle$$

and t^s is a node in state $s = \{+,-\}$. In contrast to the shared memory setting, the index-value pairs of distributed arrays now are truly distributed over the different storages of the storage tuples.

We run $t_{\max}$ instances of a program, each on its own node, enumerated from 1 to $t_{\max}$. The '+' state in the shared memory case means that the "master thread" is evaluating an expression, and all the other threads are idling. In the distributed case, the '+' state means that all the nodes are replicating the same evaluation, where in the '-' state, evaluation diverges.

To prove a statement in the distributed semantics, we build $t_{\max}$ derivation trees. Each derivation tree progresses independently of others, unless we encounter a synchronisation rule. The derivation tree on the node t modifies its local storage only, denoted $\bar{S}[t]$; the other storages of the tuple are local views of t on the storages of other nodes. At synchronisation rules, nodes synchronise their local views with respect to particular arrays to match the actual data on all nodes.

Synchronisation We introduce a rule for synchronisation across the $t_{\max}$ nodes over a pointer p. To trigger synchronisation, we introduce the auxiliary term '*sync p*'. When evaluating such a term, each node updates its local views with respect to pointer p to the actual data. Formally, we denote this as follows:

$$\textsc{Sync}\quad \frac{\bar{S}^i = \langle S^i_1, \ldots, S^i_{t_{\max}} \rangle \qquad \bar{S}'^i = \langle S^i_1 \oplus_p S^1_1(p), \ldots, S^i_{t_{\max}} \oplus_p S^{t_{\max}}_{t_{\max}}(p) \rangle}{\forall i \in \{1, \ldots, t_{\max}\} : \bar{S}^i; t^s; \rho \vdash \mathit{sync}\ p \Downarrow \bar{S}'^i;\ p}$$

This rule describes all-to-all communication of pointer p among the nodes. The assumption that makes this scheme work is that the same constants and variables on different nodes are given the same names in their local storages. This is easy to achieve, as all the nodes are executing the same program.

On a distributed system, as there is no global memory, reference counting has to be done locally. In our formalism, conceptually, each node keeps all the storages of other nodes. In practice, this is clearly infeasible, that is why we should treat the local view of storages as a read guarantee. This means that we guarantee, that during the program execution, if we will look up the actual storage of the node, we will find the value there. To maintain such a guarantee, we must synchronise before a pointer can be discarded from a local storage. We have to do this, as potentially we can have reads into the pointer of reference count zero from the other nodes.

To reflect this, we adjust the definition of the $\overset{n}{\sim}$ operation. To do this, we first define a utility function that determines whether a pointer is a distributed value or encloses distributed values.

$$\underline{\text{IsDist}}(p,S) = \begin{cases} \mathit{false} & S(p) \in \{\langle \star, V\rangle, 0, 1, \dots\} \\ \mathit{true} & S(p) = \langle D_a, V_a\rangle \\ \bigvee_{i=1}^{n} \underline{\text{IsDist}}(\rho(x_i), S) & S(p) = [\lambda x.e, \rho] \wedge \\ & \text{FV}(\lambda x.e) = x_1, \dots, x_n \end{cases}$$

Next, we introduce the distributed reference-count-altering operation $\overset{n}{\sim}_D$. As we intend to include synchronisation, which is a judgement, in principle, we would have to turn $\overset{n}{\sim}_D$ into a judgement as well. However, to simplify our notation, we will write it in the form: $(S;t^s;\rho)\ \overset{n}{\sim}_D\ p$ and it will return the updated storage. We also make sure that all the $\overset{n}{\sim}_D$ operations appear as a precondition to the final judgement. But this is already the case in the shared-memory semantics. The definition of the operation follows:

$$\frac{(\underline{\text{IsDist}}(S,p) \wedge \text{RC}\,(p,S) + n > 0) \vee \neg\underline{\text{IsDist}}(S,p)}{(S;t^s;\rho)\ \overset{n}{\sim}_D\ p = S \overset{n}{\sim} p}$$

$$\frac{\underline{\text{IsDist}}(S,p) \wedge \text{RC}\,(p,S) + n = 0 \qquad S;t^s;\rho \vdash \mathit{sync}\ p \Downarrow S';\ p}{(S;t^s;\rho)\ \overset{n}{\sim}_D\ p = S' \overset{n}{\sim} p}$$

The operation states that we do not discard pointers mapped to the distributed data from the storage, unless there are no further uses of it amongst all the nodes. For brevity in the rules, we use the $(S;t^s;\rho)\ \overset{n}{\sim}_D\ p_1 \overset{m}{\sim} p_2$ notation, which implicitly translates into:

$$S' = (S;t^+;\rho)\ \overset{n}{\sim}_D\ p_1 \wedge S'' = (S';t^s;\rho) \overset{m}{\sim} p_2$$

with S'' being the result.

Most of the rules will only affect the local storage. To keep the notation concise, we take the liberty to denote updates to the environments and reference counting using the same syntax on the storage tuples as we have used on the global storage in the previous section, *e.g.* for a given process t we write:

$$\begin{array}{rcll} \bar{S}' & = & \bar{S}, p \xmapsto{\text{rc}\,(n)} v & \text{to denote} \\ \bar{S}' & = & \langle S_1, \dots,\ S_t, p \xmapsto{\text{rc}\,(n)} v,\ \dots S_{t_{\max}}\rangle & \text{where} \\ \bar{S} & = & \langle S_1, \dots, S_{t_{\max}}\rangle & \end{array}$$

The same applies for $\bar{S} \oplus_p v$ and $(\bar{S};t^s;\rho)\ \overset{n}{\sim}_D\ p$. Which t we are modifying will be evident from the context of the rule.

We now present the rules for the distributed semantics, per convention prefixing them with D-. We start with the local constants:

D-CONST-LOCAL-1
$$\frac{v \in \{\langle \star, V_a\rangle, 0, 1, \dots\}}{\bar{S};t^s\rho \vdash v \Downarrow \bar{S}, p \xmapsto{\text{rc}\,(1)} v;\ p}$$

D-CONST-LOCAL-2
$$\frac{v = \langle D_a, V_a\rangle}{\bar{S};t^-\rho \vdash v \Downarrow \bar{S}, p \xmapsto{\text{rc}\,(1)} \langle \star, V_a\rangle;\ p}$$

For local constants, we keep all the data on every node. The same happens when a non-distributed constant is found in the sequential context (indicated by t^-). In principle, it would be possible to support distributed constants within t^-, but as any successive map/reduce will be performed locally, there is no point in doing so. Also, performance-wise, it is more efficient to ignore the distribution of the array assuming that it is $\star$ now.

When we store a distributed constant, each node stores the part that is prescribed by constant distribution. This suggests that this process requires synchronisation among the nodes, because, in order to to evaluate non-local selection, the nodes have to put the new value in their local views.

D-CONST-DIST

$$\frac{\begin{array}{c} v = \langle D_a, V_a \rangle \\ \bar{S}^1 = \bar{S}, p \xmapsto{\text{rc}\,(1)} \langle D_a, \{ i \mapsto V_a(i) \mid i \in D_a@t \} \rangle \qquad \bar{S}^1; t^+; \rho \vdash \mathit{sync}\ p \Downarrow \bar{S}^2; p \end{array}}{\bar{S}; t^+ \rho \vdash v \Downarrow \bar{S}^2;\ p}$$

Again, we assume that all the nodes allocate the same pointer for any given constant, so that it can be used as a mean of synchronisation across the nodes.

Variable lookup is the same as before. Primitive operations can be executed only on scalars, so reference counting on their arguments is always local. This means that the rule can be used, as is, in all the states of t.

As one of the very important performance optimisations, selections on distributed arrays cache the intermediate results. Consider the rule for selection:

D-SEL

$$\frac{\begin{array}{c} \bar{S}; t^s; \rho \vdash i \Downarrow \bar{S}^1;\ p_i \\ \bar{S}^1[t](p_i) = v \qquad \bar{S}^1; t^s; \rho \vdash e \Downarrow \bar{S}^2;\ p_e \qquad \bar{S}^2[t](p_e) = \langle D_a, \{ i_1 \mapsto v_1, \ldots, i_n \mapsto v_n \} \rangle \\ w = \begin{cases} v_k & \exists k \in \{1 \ldots n\} : i_k = v \\ \underline{\text{snd}}(\bar{S}^2[u](p_e))(v) & \exists u \in \{1, \ldots, t_{\max}\} : v \in D_a@u \end{cases} \\ \bar{S}^3 = (\bar{S}^2; t^s; \rho) \overset{-1}{\rightsquigarrow}_D p_e \overset{-1}{\rightsquigarrow}_D p_i \\ \bar{S}^4 = \begin{cases} \bar{S}^3 & p_e \notin \bar{S}^3[t] \\ \bar{S}^3 \oplus_p \langle D_a, \underline{\text{snd}}(\bar{S}^3[t](p_e)) \cup \{ v \mapsto w \} \rangle & p_e \in \bar{S}^3[t] \end{cases} \end{array}}{\bar{S}; t^s; \rho \vdash e\ i \Downarrow \bar{S}^4, p \xmapsto{\text{rc}\,(1)} w;\ p}$$

The completeness and the uniqueness of distributions guarantees that the case distinctions in the pre-requisites are well defined. The $\underline{\text{snd}}(\bar{S}^2[u](p_e))(v)$ expression means that we are selecting the array that binds to p_e in the storage of the node u at index v. After that, if the array is not removed due to a reference count decrease, we extend the array in our local storage with the element $v \mapsto w$. This is the implementation of caching — the next selection at the same index will be a local operation. By extending the array, we have multiple copies of the same index-value pair across different nodes. But, because arrays are read-only at this stage, *i.e.* without having introduced memory reuse, this is safe for the time being. Also, bear in mind that we can always discard the cached copies, by matching the indices of $D_a@t$ and the indices of the $\underline{\text{snd}}(\bar{S}[t](p_e))$.

Local selection is the same as in the shared memory case. The map/reduce rules follow:

D-IMAP-PAR

$$\frac{\begin{array}{c}\bar{S};t^+;\rho \vdash a \Downarrow \bar{S^1};\, p_a \qquad \bar{S^1}[t](p_a) = \langle D_a, V_a\rangle \qquad D_a@t = \{i_1,\dots,i_n\} \\ \bar{S^1};t^+;\rho \vdash f \Downarrow \bar{S^2};\, p_f \qquad \bar{S^3} = \bar{S^2} \overset{|D_a@t|-1}{\sim} p_f, p \xmapsto{\text{rc}\,(1)} \langle D_a, \{i_1 \mapsto \bot,\dots,i_n \mapsto \bot\}\rangle \\ \bar{S^3};t^-;\rho \vdash \mathit{imap}_1\ p\ D_a@t\ p_f \Downarrow \bar{S^4};\, p \\ \bar{S^4};t^+;\rho \vdash \mathit{sync}\ p \Downarrow \bar{S^5};\, p \qquad \bar{S^6} = (\bar{S^5};t^+;\rho) \overset{-1}{\sim}_D p_a\end{array}}{\bar{S};t^+;\rho \vdash \mathit{imap}\ f\ a \Downarrow \bar{S^6};\, p}$$

Here, we have replaced the union that we had in T-IMAP-PAR with the synchronisation operation. The D-IMAP-SEQversion of the rule is identical to the T-IMAP-SEQ. When we evaluate imap_1, we increase the reference count by $|D_a@t| - 1$, which gives us $|a| - t_{\max}$ references. This is because the function closures were computed locally on every node, whereas in case of shared memory, the closure was shared amongst the threads.

The distributed version of reduce looks as follows:

D-REDUCE-PAR

$$\frac{\begin{array}{c}\bar{S};t^+;\rho \vdash a \Downarrow \bar{S^1};\, p_a \qquad \bar{S^1}[t](p_a) = \langle D_a, V_a\rangle \qquad \bar{S^1};t^+;\rho \vdash f \Downarrow \bar{S^2};\, p_f \\ \bar{S^3} = \bar{S^2} \overset{|D_a@t|+t_{\max}-1}{\sim} p_f \qquad \bar{S^3}, p \xmapsto{\text{rc}\,(1)} e_{\text{neut}};t^-;\rho \vdash \mathit{reduce}_1\ p\ D_a@t\ p_f\ p_a \Downarrow \bar{S^4};\, p \\ \bar{S^5} = \bar{S^4}, p_r \xmapsto{\text{rc}\,(1)} \langle \{1 \mapsto \{1\},\dots,t_{\max} \mapsto \{t_{\max}\}\}, \{t \mapsto \bar{S^4}[t](p)\}\rangle \\ \bar{S^5} \oplus_p e_{\text{neut}};t^+;\rho \vdash \mathit{sync}\ p_r \Downarrow \bar{S^6}; p_r \\ \bar{S^6};t^-;\rho \vdash \mathit{reduce}_1\ p\ p_f\ p_r \Downarrow \bar{S^7}; p \qquad \bar{S'} = (\bar{S^7};t^+;\rho) \overset{-1}{\sim}_D p_a \overset{-1}{\sim}_D p_r\end{array}}{\bar{S};t^+;\rho \vdash \mathit{reduce}\ f\ a \Downarrow S';\, p}$$

As can be seen, similarly to the T-REDUCE-PAR we use two reduce_1 operations, one to calculate partial results, and the other to get the final value. However, in the distributed case, the array with partial values is distributed. The number of references that we increase p_f to is different from the shared-memory case because the rule assumes that every node locally recomputes the final result. Synchronisation on p_r is hidden in the reference-count-update operation.

Theorem 4. *The distributed semantics evaluates the same values as the shared-memory one.*

$$\frac{\emptyset;t^+;\emptyset \vdash e \Downarrow_T S_1;\, p_1 \qquad \forall u \in \{1,\dots,t_{max}\} : \bar{\emptyset};u^+;\emptyset \vdash e \Downarrow_D \bar{S^u};\, p_u}{S_1(p_1) = \left(\bigcup_{u=1}^{t_{max}} \bar{S^u}[u]\right)(p_u)}$$

where union joins distributed values of the environment according to its distribution and picks any of the local values, as they are the same.

Proof sketch. We show that, after any evaluation step in the distributed semantics, the union of local storages is the same as the shared storage after the corresponding step in the shared-memory semantics. □

ZU064-05-FPR paper 17 March 2020 16:57

5.1 Reuse under distributed semantics

Under the distributed semantics, we can reuse arrays in the same way as in the abstract and shared-memory ones. There are two important differences, though. First, we must ensure that the reuse criteria is applied for the overall index space of the distributed array.

Second, at the synchronisation, we must discard all previously cached values that may have appeared in the array during selections. This needs doing because if the array will be successively reused, the cached values will not be updated. To illustrate this, consider a situation in which array *a* during the first *imap* was reused, and the value at index 1 was cached on every node. Assuming that the second reuse of *a* updates the element at index 1, all the cached copies become invalid. In the D-IMAP-PAR-R, this happens implicitly. We omit the D-IMAP-PAR-Rrule here to save some space.

Theorem 5. *Reuse criteria can be applied in the distributed semantics.*

Proof sketch. By showing the identity of the evaluated values for every index and demonstrating that none of the cached values are preserved. □

5.2 Implementation considerations

We now discuss extensions of distributed λ_{DP}, and give some implementation details.

In this paper, we have introduced distributions as a form of dynamic typing. In principle, distributions can be introduced as static types; we could even track, in the distribution-type system, the level of the nested map/reduce construct which would run in parallel.

Synchronisation of reference counting can be implemented as follows: we choose a node, where we will keep global reference counts for every distributed array. A global reference count is a number which is initialised at the allocation of an array to the number of nodes. At runtime, when a local reference count of the pointer *p* decreases to zero, the node checks the global reference count. If the value is greater than one, it decreases this value by one, and carries on with the computation without further waiting. If the value is one, then the node decreases it by one and initiates removal of *p*s on every node. In this way, our garbage collection becomes more dynamic, and the amount of time that nodes have to wait because of synchronisation drops significantly.

One of the most important advantages of DSMs is their ability to implement fetches of non-local elements transparently via reacting on page faults and using MMUs to map pages in the given address space. Our current distributed semantics for non-local selections communicates individual values of the array. In reality, it would make more sense to communicate pages and use MMU to map those pages into the right address space. By doing this, formally we might end-up in a situation when the cached page contains elements with undefined values. This may happen when we evaluate an *imap* with reuse, and obtain a page that amongst the read-only elements contains the elements to be modified. When we get the page, we do not know whether the value has been written or not. However, since we keep distributions based on individual indices, our reuse criteria guarantees that undefined values will be never read during the evaluation of *imap*. At the end of the *imap*, the cached pages will be released. Therefore, our formalism can be straightforwardly extended to cache pages on non-local selections.

Although our formalism relies on the fact that work and data distributions coincide, this need not be the case. If one wants to reduce the amount of data communications, then it is easy to construct an example in which tightly coupled work and data distributions would create more non-local data reads than in the case when work and data distributions are disjoint and at the end of the operation the computed result is sent to the owner. To accommodate this, we could introduce notions of work and data distributions separately, or we could envision redistribution of the evaluated results.

6 Related Work

We identify three main areas of related work: semantics and reference counting, parallel languages and distributed shared memory.

Semantics and reference counting An alternative approach to step-wise refinements of semantics is to provide compilation schemes for transforming the source language into a target language in which some of the properties of the underlying architecture are made explicit. In a way, one could think of this work as describing a compiler, whereas our paper is more oriented towards describing a runtime system.

For example, in (Pierce, 2004)[Chapter 1], the formalism introduces a reference-counting scheme based on linear λ-calculus; but the overall scheme requires reference-counting operations like *incref* or *decref* to be explicit.

In (Tofte & Talpin, 1994), the authors introduce the concept of stack of regions, which evokes the concept of our storages. The paper describes a compilation scheme from the Milner's call-by-value λ-calculus into the language in which regions are made explicit, and demonstrates that the translation scheme is correct.

In (Hudak, 1986), the author introduces reference counting, providing a denotational semantics of an applicative first-order functional language. Basic reference-counting operations are performed in the same way that we describe here; the paper provides a proof of correctness of their reference-counting scheme.

Parallel languages In the functional world, support for distributed systems has existed for a very long time. One of the strongest features of Erlang (Armstrong, 2007) is its ability to support distributed systems in a reliable way. Haskell also provides a number of solutions to support distributed systems. Eden (Loogen *et al.*, 2005) extends Haskell to support distributed memory parallelism, providing process abstractions similarly to λ-abstractions. CloudHaskel (Epstein *et al.*, 2011) supports explicit process abstractions and message passing; and HdPH (Maier & Trinder, 2011) supports a fork-like operator called spark to initiate computation on a remote node.

Distributed shared memory Classical work on distributed shared memory is known to be difficult, as it aims to deal with arbitrary writes between two memory synchronisations, requiring a DSM to keep track of the writes done on each node and to exchange this information upon synchronisation, e.g. by exchanging page diffs (Carter *et al.*, 1991). Such an exchange introduces a lot of overhead which is can be reduced significantly with the idea of data ownership as for example implemented in Cachemere (Kontothanassis *et al.*, 2005). In the functional setting the situation is even better. Our distributed-memory semantics enables to control which thread is writing which values which effectively ensures that writes exclusively are done by the owner, erasing the communication overhead upon synchronisation completely.

An alternative approach to DSMs, Partitioned Global Address Space model (PGAS) (Coarfa *et al.*, 2005), has recently attention, as several languages, including Chapel (Chamberlain *et al.*, 2007), X10 (Charles *et al.*, 2005), UPC (Carlson *et al.*, 1999) are built upon a PGAS model. The main idea of PGAS is to introduce locality awareness on top of message passing. PGAS presents distributed memory as a continuous address space. However, the address space is also logically partitioned among threads or processes, which allows local computations. nodesAs a consequence *every* selection into a distributed array has to distinguish whether this is a local operation or a global one. This distinction cannot be hidden in the hardware as it's done in the DSM approach. Instead, the software has to capture this either through programmer specification, as is done in UPC, or by means of a combination of program analysis and dynamic checks, as it is done in Chapel.

7 Conclusions and future work

In this paper, we present three successive refinements of the operational semantics of a functional language for arrays and array-oriented data-parallel constructs. The refinements introduce storages, reference counting, an analysis to perform destructive updates, the concept of distributions and threads and the final refinement made it possible to reason about distributed evaluation on cluster-like systems. To our knowledge, this is the first time that an operational semantics specifically for data-parallelism is defined and, at the same time, refined to capture implementation details for sequential and parallel executions alike. The presented formalism is concise yet powerful enough to reason about subtleties of the underlying implementation, even in the context of a distributed memory implementation on a cluster.

As it turns out, several subtleties of pre-existing implementations in the context of handling storage do not only become more evident, they actually can be shown to be necessary and sufficient to abide the original semantics. One example to this effect is the handling of reference counts during parallel execution on shared memory systems. Another example is the owner-compute observation in the distributed memory case which enables a very low-overhead DSM implementation: all synchronisations that happen on barriers can be implemented with no data exchange and selections into globally distributed arrays do come with no software overhead involved. This starkly contrast with more generic DSM settings where there is a necessity to track write operations and exchange these upon synchronisation.

These observations form a clear vision of future work. First, we wish to verify that an implementation of the DSM that includes the above-mentioned properties provides, in practice, sufficient parallel performance. After that, the formalism that we have presented so far can be substantially extended. Specifically, we would like to capture conditions and recursive functions. Arrays can be either nested or of a higher rank or both. The *imap* construct can be extended to make reuse analysis much more straightforward. It would be very interesting to relax the tight coupling between the work and data distributions, and determine what a distribution for a nested or multi-dimensional arrays might look like. Finally, exploring trade-offs of running multiple threads on each cluster node is of interest.

References

Allen, J. R., Kennedy, Ken, Porterfield, Carrie, & Warren, Joe. (1983). Conversion of control dependence to data dependence. *Pages 177–189 of: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '83. New York, NY, USA: ACM.

Armstrong, Joe. (2007). *Programming erlang: Software for a concurrent world*. Pragmatic Bookshelf.

Axelsson, Emil, Claessen, Koen, Sheeran, Mary, Svenningsson, Josef, Engdal, David, & Persson, Anders. (2011). *Ifl 2010, alphen aan den rijn, the netherlands, september 1-3, 2010, revised selected papers*. Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. The Design and Implementation of Feldspar, pages 121–136.

Blelloch, G. E., & Sabot, G. W. 1988 (Oct). Compiling collection-oriented languages onto massively parallel computers. *Pages 575–585 of: Frontiers of Massively Parallel Computation, 1988. Proceedings., 2nd Symposium on the Frontiers of.*

Blelloch, Guy E. (1992). *Nesl: A nested data-parallel language*. Tech. rept. Carnegie Mellon University, Pittsburgh, PA, USA.

Carlson, William W., Draper, Jesse M., Culler, David, Yelick, Kathy, Brooks, Eugene, & Warren, Karren. 1999 (May). *Introduction to UPC and language specification*. Tech. rept. CCS-TR-99-157. Center for Computing Sciences, IDA, Bowie, MD.

Carter, John B., Bennett, John K., & Zwaenepoel, Willy. (1991). Implementation and performance of munin. *Pages 152–164 of: Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. SOSP '91. New York, NY, USA: ACM.

Chakravarty, Manuel MT, Leshchinskiy, Roman, Peyton Jones, Simon, Keller, Gabriele, & Marlow, Simon. (2007). Data parallel haskell: a status report. *Pages 10–18 of: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*. ACM.

Chakravarty, Manuel M.T., Keller, Gabriele, Lee, Sean, McDonell, Trevor L., & Grover, Vinod. (2011). Accelerating haskell array codes with multicore gpus. *Pages 3–14 of: Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*. DAMP '11. New York, NY, USA: ACM.

Chamberlain, B.L., Callahan, D., & Zima, H.P. (2007). Parallel programmability and the Chapel language. *Int. j. high perform. comput. appl.*, **21**(3), 291–312.

Charguéraud, Arthur. (2013). Pretty-big-step semantics. *Pages 41–60 of:* Felleisen, Matthias, & Gardner, Philippa (eds), *Programming Languages and Systems*. Lecture Notes in Computer Science, vol. 7792. Springer Berlin Heidelberg.

Charles, Philippe, Grothoff, Christian, Saraswat, Vijay, Donawa, Christopher, Kielstra, Allan, Ebcioglu, Kemal, von Praun, Christoph, & Sarkar, Vivek. (2005). X10: An object-oriented approach to non-uniform cluster computing. *Sigplan not.*, **40**(10), 519–538.

Chiw, Charisee, Kindlmann, Gordon, Reppy, John, Samuels, Lamont, & Seltzer, Nick. (2012). Diderot: A parallel dsl for image analysis and visualization. *Sigplan not.*, **47**(6), 111–120.

Coarfa, Cristian, Dotsenko, Yuri, Mellor-Crummey, John, Cantonnet, François, El-Ghazawi, Tarek, Mohanti, Ashrujit, Yao, Yiyi, & Chavarría-Miranda, Daniel. (2005). An evaluation of global address space languages: Co-array Fortran and Unified Parallel C. *Pages 36–47 of: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. New York, NY, USA: ACM.

Epstein, Jeff, Black, Andrew P., & Peyton-Jones, Simon. (2011). Towards haskell in the cloud. *Pages 118–129 of: Proceedings of the 4th ACM Symposium on Haskell*. Haskell '11. New York, NY, USA: ACM.

Feautrier, Paul. (1991). Dataflow analysis of array and scalar references. *International journal of parallel programming*, **20**(1), 23–53.

Feo, John, Cann, David C., & Oldehoeft, R. R. (1990). A report on the sisal language project. *J. parallel distrib. comput.*, **10**(4), 349–366.

Grelck, Clemens, & Scholz, Sven-Bodo. (2006). SAC – A Functional Array Language for Efficient Multi-threaded Execution. *International journal of parallel programming*, **34**(4), 383–427.

Hauck, E. A., & Dent, B. A. (1968). Burroughs' b6500/b7500 stack mechanism. *Pages 245–251 of: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). New York, NY, USA: ACM.

Henriksen, Troels, Elsman, Martin, & Oancea, Cosmin E. (2014). Size slicing: A hybrid approach to size inference in futhark. *Pages 31–42 of: Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*. FHPC '14. New York, NY, USA: ACM.

Hudak, Paul. (1986). A semantic model of reference counting and its abstraction (detailed summary). *Pages 351–363 of: Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP '86. New York, NY, USA: ACM.

Hudak, Paul, & Bloss, Adrienne G. (1985). The aggregate update problem in functional programming systems. *Pages 300–314 of: Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*.

Kahn, G. (1987). Natural semantics. *Pages 22–39 of:* Brandenburg, FranzJ., Vidal-Naquet, Guy, & Wirsing, Martin (eds), *STACS 87*. Lecture Notes in Computer Science, vol. 247. Springer Berlin Heidelberg.

Kennedy, Ken, & Allen, John R. (2002). *Optimizing compilers for modern architectures: A dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Kontothanassis, Leonidas, Stets, Robert, Hunt, Galen, Rencuzogullari, Umit, Altekar, Gautam, Dwarkadas, Sandhya, & Scott, Michael L. (2005). Shared memory computing on clusters with symmetric multiprocessors and system area networks. *Acm trans. comput. syst.*, **23**(3), 301–335.

Leroy, Xavier. (2009). Formal verification of a realistic compiler. *Communications of the acm*, **52**(7), 107–115.

Loogen, Rita, Ortega-mallén, Yolanda, & Peña marí, Ricardo. (2005). Parallel functional programming in eden. *J. funct. program.*, **15**(3), 431–475.

Maier, Patrick, & Trinder, Philip W. (2011). Implementing a high-level distributed-memory parallel haskell in haskell. *Pages 35–50 of:* Gill, Andy, & Hage, Jurriaan (eds), *IFL*. Lecture Notes in Computer Science, vol. 7257. Springer.

Pierce, Benjamin C. (2004). *Advanced topics in types and programming languages*. The MIT Press.

Pouchet, L. N., Bastoul, C., Cohen, A., & Vasilache, N. 2007 (March). Iterative optimization in the polyhedral model: Part i, one-dimensional time. *Pages 144–156 of: Code Generation and Optimization, 2007. CGO '07. International Symposium on*.

Svensson, Joel, Claessen, Koen, & Sheeran, Mary. (2010). Gpgpu kernel implementation and refinement using obsidian. *Pages 2065–2074 of:* Sloot, Peter M. A., van Albada, G. Dick, & Dongarra, Jack (eds), *ICCS*. Procedia Computer Science, vol. 1. Elsevier.

Tofte, Mads, & Talpin, Jean-Pierre. (1994). Implementation of the typed call-by-value λ-calculus using a stack of regions. *Pages 188–201 of: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. New York, NY, USA: ACM.